Audio Plugin Development with QtGrpc

A Journey Into Remote GUIs

Dennis Oberst

19 October 2023

Abstract

The Qt project is a powerful and versatile C++ framework widely used for developing cross-platform applications. Additionally, Qt is no stranger to the audio-industry, with a strong presence within professional audio companies. If we look into the expanding world of audio-plugins however, we would steer into the void when looking for Qt user-interfaces. With this work I will explain the reasons and problems that led to this state and show a new and innovative approach in solving the problems.

Contents

Chapter 1: Introduction

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

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

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

> Code once, deploy everywhere.

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

A short query of:

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

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

1.1 Background

1.1.1 Plugins: Shared Libraries

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

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

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

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

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

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

Shared libraries can be further categorized into:

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

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

Since shared libraries are at the foundation of every plugin, it comes beneficial to explore them a bit deeper by going through a minimal example:

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

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

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

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

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

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

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

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

int main()
{
    // Load the shared library.
    void* handle = dlopen("build/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 `sharedlibhost` executable.

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

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

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

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

./build/simplehost
```

And finally run our script:

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

1.1.1 Plugins: Overview and Significance

Plug-*ins* in their most basic form extend the functionality of a plugin-loading-host dynamically. We could think of them as functionality that is made available *on-demand*. They are used all around the software and hardware world and can be found in a multitude of areas. Be it the extension of specialized filters for image processing applications like *Adobe Photoshop*, dynamically loadable drivers for operating systems like *GNU/Linux* (Vandevoorde, 2006) or operating system dependent features as used in the Qt framework.

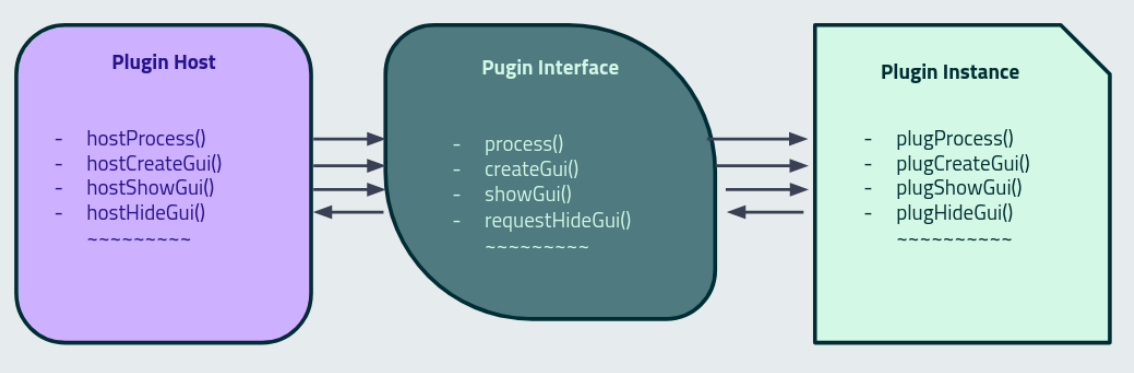

Figure 1: basic plugin architecture

Figure 1 describes this process. After successfully loading

- Define what plugins are and their role in software systems.
- Explain how plugins enhance the functionality and extensibility of software applications.
- Discuss the advantages of using plugins in the audio processing domain and its impact on the user experience.

1.2 Problem Statement

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

- QApplication, static environment, can
- Shared Library - non blocking

1.3 Objectives

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

1.4 Scope and Limitations

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

Chapter 2: The Qt Framework

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

2.2 Qt Modules and Architecture, QtGRPC

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

2.3 Qt APIs and Tools

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

Chapter 3: The Clap Audio Plugin Format

3.1 Introduction to Clap

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

3.2 Clap Plugin Structure

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

3.3 Clap API and Specifications

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

Chapter 3.5: Realtime, Events, Data Structures, Communications

Chapter 4: Integrating Qt with Clap

4.1 Design Considerations

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

4.2 Architecture and Implementation

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

Chapter 5: Experimental Results and Evaluation

5.1 Test Setup and Methodology

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

5.2 Results and Analysis

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

Chapter 6: Discussion and Conclusion

6.1 Summary of Findings

- Summarize the key findings and outcomes of the research.

6.2 Discussion of Results

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

6.3 Contributions and Future Work

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

Bibliography

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