C++ Desktop Application Architecture for Digital Amplifier Connectivity

Hayden Bursk · Follow

Published in Fender Engineering · 14 min read · Mar 19, 2020

Tone Desktop for Mustang LT

Fender has been making amplifiers since the 1940s, and modeling amps since the mid 2000s. In 2017 we launched the Mustang GT and a companion mobile app, Fender Tone, for controlling it via BLE. Fast forward 3 years and we are ready to release Tone 3.0, which introduces support for the new Mustang GTX, Android tablet and iPad support, and an entirely new skeuomorphic UI. Going from a flat UI to a modern skeuomorphic one creates a number of design and engineering challenges. But why make the switch from a flat UI? Fender is uniquely positioned to be one of the few guitar companies that makes and owns the brand of the tube amplifiers we are modeling in our digital amps. Giving our customers the visual sense as well as auditory experience of playing through a Twin Reverb just makes sense. But boy, that's a lot of bitmaps.

Aside from the challenges of managing hundreds of bitmaps, figuring out new control layout systems, and displaying detailed signal chains on a small screen in two existing native codebases for iOS and Android, there was an opportunity. We built an entirely new app for two new platforms. We built Tone Desktop for Mac and Windows to support Mustang LT and Rumble LT over USB.

As a software developer at a larger company, or even a smaller one, there's not too many opportunities to build a new app from the ground up — to use your experience and expertise to start fresh. This post is a short review of some of the highlights from the project.

What we did well

To start, we examined what we wanted to keep and what we wanted to improve from the original Tone to Mustang GT communication. First the good stuff.

Pick a source of truth

This tenet is an excellent beacon in guiding us to how our API should be designed, sync between the software and the hardware should work, and who wins if there's a discrepancy. In our case, the hardware (amp) was the source of truth and the app was the companion.

Messaging Protocols

All our messages are built on protobuf. It creates small, optimized messages. The messages can be generated into platform specific classes and dealt with as objects. And we had an existing library of messaging already defined.

Team Collaboration

The amp hardware team and the Tone team are separated across state lines, and work quite independently of each other. However, the most successful additions to the original Tone app, like Rumble support and Backup and Restore, were through tight collaboration between teams. We needed more of that.

What we can do better

Not everything was perfect though. There were certainly areas we could improve on.

The Parameter Feedback Problem

Tone uses a Protobuf messaging library called FenderMessage to communicate with the amp. The amp uses the same messaging library internally to communicate between the control panel, application layer, and DSP. When a physical knob was turned on the amp, it would broadcast a SetParameter message to Tone. However, the layer that broadcasted the message didn't know the source of the change. When Tone wanted to change a parameter, Tone would also send a *SetParameter* message to the amp. The amp would then echo this back to Tone. Since the user was dragging a knob on Tone, this echoed message would be old and listening to it would cause Tone's controls to jump back in time, and stutter. A system to ignore *SetParameter* messages from the amp while Tone's controls were changing was put into place, but eventually the echo from the amp was turned off in later firmware versions.

Our new system would need a way to identify the source of the *SetParameter* message to avoid this feedback problem but still receive it to maintain acknowledgments and sync between Tone and the amp.

Stateful Protocol

The problem with a stateful protocol is that the same message would have to be treated differently depending on what actions or messages preceded it. This made for very fragile application logic that could easily break if the messaging API changed or new features needed to be added. We didn't want that. A *LoadPreset* message should always load a preset the same exact way. A *SavePreset* should always save it. That gives us a solid testable deterministic system. We needed a Stateless Protocol.

The Approach

The API and communication layers were the main priority. The hardware team's deadline for a code freeze would be well before the software team, so we had to design it to work with an app we hadn't yet made, and make sure everything was solid and well tested in the process. The Tone team had a brain-storm of every use case we could think of on how someone would use Tone with an amp. We ended up with a nice collection of post-its that were then transferred into a Google Doc and shared with the hardware team. Every use case would be documented with a corresponding message, the data the message contained, and the response. In some situations, there might be multiple responses from one request. Sending a parameter change would result in both a parameter changed and a preset edited message. In some cases, a different use case had the same messaging pattern. Saving a preset to a new slot, and duplicating an existing preset were quite similar while being contextually different to the user. This approach of thinking of use cases first helped ensure we didn't miss anything much.

The Architecture

The architecture was thought of before the API was designed, and influenced API decisions with buy-in from the hardware team. While the entire client layer was abstracted into its own repo, **tone-sdk**, unit tested and thoroughly vetted, it wasn't until we had a knob on screen turning in sync with a

physical hardware knob on the amp that we knew our proposed system was going to work out.

Our architecture was built on a simple premise. Tone would have its own data model representing the state of the amp. The UI would send messages to the amp. The amp would respond. The model would update its state. The UI would be notified of any changes. The UI would never ask the model for any state directly. It would just update based on asynchronous changes.

Oh, and it should be lock-free. I don't advocate for lock-free programming as a blanket approach to all multithreaded applications, but for our use and approach, it made sense. First, any locks we would have used would protect making requests and accessing the data from the main thread. That would mean blocking the main thread, thus freezing up the UI. Second, I use the term lock-free in reference to our main request and read use case. You will see code examples below that use `std::lock_guard` as well as `std::condition_variable` for various threading needs.

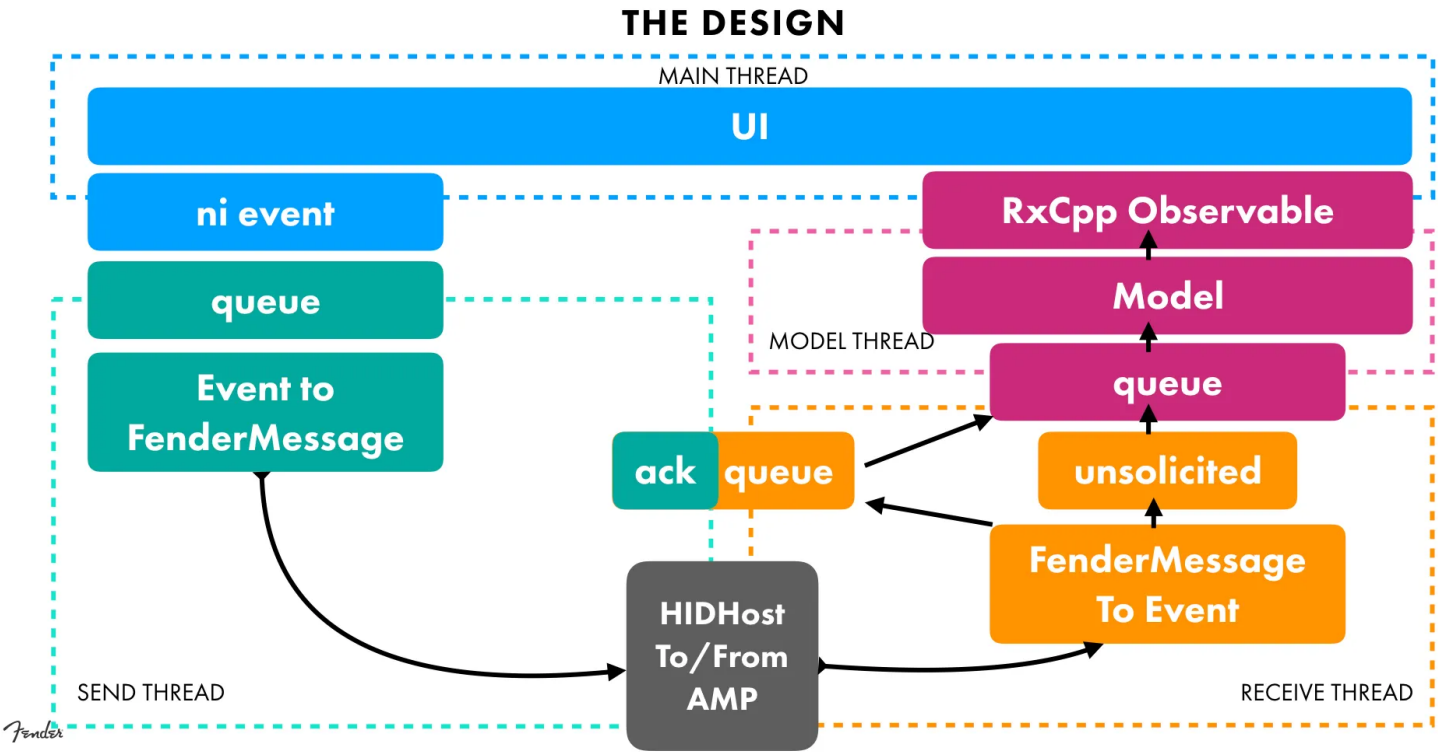
Our Model

Before we talk about threading, let's talk about the model. Why not just query the amp for its state all the time? Well, we would have loved to, but bandwidth is a problem. Over USB HID, it takes a couple seconds to sync all the presets on the amp. If that process was on the order of milliseconds, it might have been feasible. But once we leave USB and consider BLE, transfer rates drop and there's no way we'd get the full state as fast as we need it. Also, the amp API is transactional. We learn about one change at a time, and we need to keep a record of those changes to make sure we're in sync with the amp. We ended up writing several nested classes to represent the state of the signal chain, its nodes, parameters, and all the preset slots on the amp. Reusing the same code the amp used would have been great, but the amp's

data model was too tightly coupled to the DSP code. Recreating all that logic from scratch was easier. Changing parameters, swapping effects, loading and saving presets. All that logic was rewritten for use with our new Tone architecture. Most of our unit tests are around validating model state.

Threading

Here's a diagram I show to new software devs during on-boarding.



They usually say, "Makes sense." I'd like to believe them. The message flow is counter-clockwise in this diagram with three threads plus the main thread in play. Some interaction happens from the UI and a corresponding event object is created and funneled to a lock-free queue. We use the ni::matchine library from Native Instruments for event types and the boost::lockfree::queue. When an event is placed in the send thread queue, a `std::condition_variable` is notified and the thread begins processing the queue. Yes, technically a condition variable uses a `std::unique_lock`, but

honestly, how else would you do it? We still don't have any `std::lock_guard` wraps around data access. That event is converted to a protobuf FenderMessage and sent to the amp over HID.

Let's pause a second. Why not just use FenderMessage directly? It's another layer of abstraction so that this same architecture could be used for different messaging protocols. Let's say we decide to change to another format other than protobuf. We'd just need a new translation layer.

The HID layer is also abstracted to be a transport layer. In the same way as messaging, if the transport mechanism was to change, to BLE for example, we would only need a new transport layer, and everything would continue to work as expected.

Now that we've sent a message to the amp, the send thread has to wait. The contract between client and the Mustang LT amp is that we only send one message at a time, and wait for a response before sending another. That restriction is handled at the tone-sdk level, and the high-level client app has no knowledge of the inner workings. We do eventually time out on waiting. Otherwise, never receiving a message back from the amp would block us indefinitely.

The amp will send back one or more messages related to our request. If you recall, in the case of our first parameter change, we receive the parameter changed event as well as a preset edited event. The send thread can't continue sending messages until all the responses have been received, so messages from the amp are tagged as either an acknowledgement, a response, or an unsolicited message. In case of a parameter change, we would receive one response for the parameter change, and then the last message, the preset edited status, would be the acknowledgement. The

response and the acknowledgement are converted back into our custom event type, grouped together, and then funneled to the acknowledgement queue, which is what the send thread has been waiting for. Once the send thread sees those messages, it funnels them to the model queue and continues sending any remaining messages in its queue.

The model queue is the only access to the data model. This is how we avoid any locks for write access. The model thread will be notified when there's an available event and it will use `ni::match` and fire off the appropriate lambda to handle the event. In this example, the parameter value will be updated and the preset edited flag will be set to true.

The model then uses RxCpp to notify the UI. We'll discuss that in just a moment.

There's one other part of the thread handling I want to call out which is unsolicited messages. These are messages generated directly from the amp. Really, at anytime, the user can reach over and turn a knob on the amp, changing a preset or a knob, and the amp will respond as fast as it can. This produces an unsolicited message and Tone will process those as fast as it can. Unsolicited messages bypass the acknowledgment queue, because they are not an acknowledgement to any pending request, and are sent to the model queue right away. If an unsolicited message were to arrive in the middle of a grouping of messages from a Tone request, the grouping will stop, the pending messages will be fired off the model, then the unsolicited message will be fired off, so order is maintained. Then, the acknowledgement queue will resume batching until the entire acknowledgement group has been received.

We batch those messages so that updates to the UI will happen in a visually synchronized way. Otherwise, we'd see the preset status indicator update just after a parameter change rather than in lock-step. All the updates happening together in one frame of drawing feels more solid to the user.

RxCpp

RxCpp is the Reactive Extensions library for C++ for values-distributed-in-time. Early on in the process, we had our eye on using RxCpp for notifying the UI of any model changes. The prospect of being able to chain observables while modifying their values was very enticing. However, we really only ended up using some very basic features of the library. Subjects were either of the `Rx::behavior<T>` type or the `Rx::subject<T>` type. The difference between the two is: do you want to your subscription lambda to be called as soon as it's set, or not until it changes next time? For the signal chain and the preset edited status, we chose `Rx::behavior` in order to be notified of the current state as soon as the application subscribed. But for a parameter change, we used `Rx::subject`, since it did not have a state to provide, but just the notification of a future change.

In order to keep the application code from being littered with RxCpp objects, we wrapped the implementation into a ModelObservable class.

ModelObservable.h

```
#pragma once

#include <rxcpp/rx.hpp>

#include <functional>
#include <memory>
#include <mutex>
```

```
#include <unordered_map>
#include <vector>

namespace Rx
{
using namespace rxcpp;
using namespace rxcpp::sources;
using namespace rxcpp::operators;
using namespace rxcpp::subjects;
using namespace rxcpp::util;
using namespace rxcpp::schedulers;
} // namespace Rx

namespace fmic::tone::model
{
class ModelObservable
{
public:
    using SubscriberVec = std::vector<Rx::composite_subscription>;
    using SubscriberMap = std::unordered_map<void*, SubscriberVec>;

    ModelObservable()          = default;

    void unsubscribe( void* subscriber );

protected:
    void addSubscriber( void* subscriber,
                        Rx::composite_subscription&& subscription );

    const static std::function<void( std::exception_ptr )>
    on_error_rethrow;

template <typename T, typename Callback>
auto subscribe( T&& rxItem, Callback&& cb )
{
    return rxItem.get_observable().subscribe(
                  std::forward<Callback>( cb ),
                  on_error_rethrow );
}

private:
    SubscriberMap m_subscribers;
    std::mutex    m_subscribersLock;
};
} // namespace fmic::tone::model
```

ModelObservable.cpp

```
#include "ModelObservable.h"

using namespace fmic::tone::model;

const std::function<void( std::exception_ptr )>
ModelObservable::on_error_rethrow = []( std::exception_ptr p ){
    std::rethrow_exception( p );
};

void ModelObservable::unsubscribe( void* subscriber )
{
    std::lock_guard<std::mutex> lockGuard( m_subscribersLock );
    if ( m_subscribers.count( subscriber ) == 0 )
    {
        return;
    }
    else
    {
        auto& subscribers = m_subscribers[subscriber];
        for ( const auto& sub : subscribers )
        {
            sub.unsubscribe();
        }
        subscribers.clear();
    }
}

// ----------------------------------------
// Private
// ----------------------------------------

void ModelObservable::addSubscriber( void* subscriber,
Rx::composite_subscription&& subscription )
{
    std::lock_guard<std::mutex> lockGuard( m_subscribersLock );
    if ( m_subscribers.count( subscriber ) == 0 )
    {
        m_subscribers[subscriber] =
        ModelObservable::SubscriberVec();
    }

    m_subscribers[subscriber].push_back( subscription );
}
```

The trick is using the memory address of the subscriber as the key to hold onto the subscription. For the application, it's as simple as passing this when calling addSubscriber and unsubscribe. A ModelObservable subclass would

then have RxCpp members, and subscription and notification methods. For example:

```
// – – – – – – – – – – – – – – – – – – – – –
// Preset Dirty
// – – – – – – – – – – – – – – – – – – – – –

void ToneDeviceModelObservable::subscribePresetEditedStatus( void*
subscriber, std::function<void( bool )> callback )
{
    addSubscriber( subscriber,
                   subscribe( m_rxPresetEditedStatus, std::move(
                       callback ) ) );
}

void ToneDeviceModelObservable::notifyPresetEditedStatus( bool
isEdited )
{
    m_rxPresetEditedStatus.get_subscriber().on_next( isEdited );
}
```

The application would call `subscribePresetEditedStatus` from as many places as it needed, but the `notifyPresetEditedStatus` would come from within tone-sdk when the model state had changed. There is a `std::lock_guard` in the code to protect the subscriber map, but internally, RxCpp is thread safe.

Some pros and cons on this wrapper approach.

- PRO — The RxCpp API is a bit verbose, and this abstracts that away.
- PRO — If we changed libraries, we're already abstracted.
- PRO — Using *this* as a token avoids duplicate subscriptions.
- CON — Using *this* as a token is a bit arbitrary.
- CON — Explicit disconnection is required vs an RAII approach where the client would need to hold onto the `Rx::subscription` object and

unsubscribe on destruction.

A Note About Linearizability

We had originally implemented our messaging queues using the `moodycamel::ConcurrentQueue`. It was fast and allowed multiple producers and consumers. At the time, we weren't so clear on the architecture that we could commit to a single producer/single consumer queue. In fact, there are various instances where a queue has two producers, so a multi queue is in fact required. However, one very important fact we overlooked on the `moodycamel::ConcurrentQueue` is called out in his section "Reasons not to use".

> *"My queue is **not linearizable**... The foundations of its design assume that producers are independent; if this is not the case, and your producers co-ordinate amongst themselves in some fashion, be aware that the elements won't necessarily come out of the queue in the same order they were put in relative to the ordering formed by that co-ordination (but they will still come out in the order they were put in by any individual producer)."*

So, if you use the queue from a single thread, everything you put in it stays in order. But if you add to it from another thread, the new events will insert themselves in between the existing events. This was discovered when we added our Backup and Restore feature. The restore process spawned a new thread to parse a backup file into a series of messages to send the amp. During unit testing, these messages were splicing themselves in between the normal sync messages on connection, even though we had waited for all the sync messages to get queued up.

As a result, we decided to abandon the `moodycamel::ConcurrentQueue` and replace the queues with the `boost::lockfree::queue`. The performance for

our low thread count was comparable to the moodycamel queue. There was one hiccup in the change though, which was that objects in a `boost::lockfree::queue` must be trivially destructible, (use raw pointers), and we had been using `std::shared_ptr<Event>` of our `ni::type_heirarchy` Event type. We decided to use `std::unique_ptr<Event>` instead and created this class to handle memory management:

```
#pragma once

#include <boost/lockfree/policies.hpp>
#include <boost/lockfree/queue.hpp>

#include <memory>
#include <stdexcept>

namespace fmic
{

/**
* Multi-producer, Multi-consumer, threadsafe, linearizable queue.
* @discussion
* This class wraps boost::lockfree::queue which can only support
* trivially destructable items.
* This wrapper enforces ownership semantics of the items place
* inside of it.
* They are owned by the queue, until popped, then ownership is
* transferred to the caller.
*/
template <typename Item, size_t Capacity = 256>
class ConcurrentQueue
{
private:
    boost::lockfree::queue<Item*,
                        boost::lockfree::capacity<Capacity>> m_queue;

public:
    ConcurrentQueue() = default;
    ~ConcurrentQueue()
    {
        while ( auto garbage = pop() ) {}
    }

/**
* Creates an item and place it on the back of the queue
*/
template <typename ItemSubclass, typename… Args> void push( Args…
args )
```

```
{
    m_queue.push( new ItemSubclass( args… ) );
}
```



```
{
    if ( item == nullptr )
    {
        throw std::invalid_argument( “null items may not be placed
                                      in the queue” );
    }
    m_queue.push( item.release() );
}

/**
* dequeues the item at the front of the queue.
* @return nullptr if the queue is empty
*/
std::unique_ptr<Item> pop()
{
    Item* i = nullptr; if ( m_queue.pop( i ) )
    {
        return std::unique_ptr<Item>( i );
    }
    return nullptr; }
};
} // namespace fmic
```

The Application

In the end, we’re very happy with our new architecture. It has proven to be quite robust and performant. Its goal was to provide a solid foundation for an excellent desktop app we hope all Mustang LT and Rumble LT owners enjoy using. In a future post, we’ll discuss the high level application framework and our use of MVVM.

Here’s a fun gif of turning the knobs on the amp and Tone’s response. Enjoy!

Special thanks to Kevin Dixon and Alex Wood for technical editing and contributions on this post.

Application Development | Amplifier | Multithreading | Desktop App | Pro Audio

Written by Hayden Bursk

Follow

11 Followers · Writer for Fender Engineering

Mobile Engineering Manager at Fender Musical Instruments

More from Hayden Bursk and Fender Engineering

Liam Crawshaw in Fender Engineering

Stacking elements with CSS Grid

A guide to stacking DOM elements with z-index, Grid, and React

8 min read · Mar 28, 2023

3

Michael Garski in Fender Engineering

Passing the Torch

Prior to taking on the responsibilities of leading a team at Fender, I worked as a...

3 min read · May 23, 2023

4 1

KP Krishnamoorthy in Fender Engineering

Terraforming legacy infrastructure

Around a year ago, Fender acquired PreSonus, one of the leaders in audio...

7 min read · Nov 28, 2022

Michael Garski in Fender Engineering

TBT: The CUBE at Serverlessconf

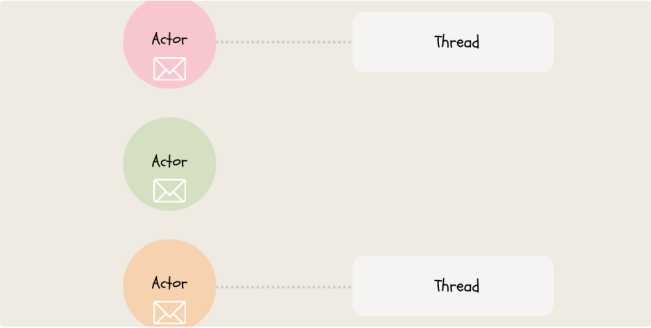
While I was at Serverlessconf San Francisco in July of 2018, I had a chance to sit down with...

1 min read · Apr 18, 2019

3 1

See all from Hayden Bursk See all from Fender Engineering

Recommended from Medium



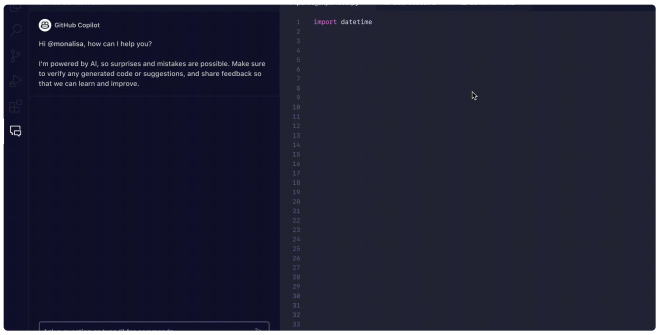
Nidhey Indurkar

How did PayPal handle a billion daily transactions with eight virtu...

I recently came across a reddit post that caught my attention: ‘How PayPal Scaled to...

7 min read · Jan 1

3.6K 36



Jacob Bennett in Level Up Coding

The 5 paid subscriptions I actually use in 2024 as a software engineer

Tools I use that are cheaper than Netflix

· 5 min read · Jan 4

5K 58

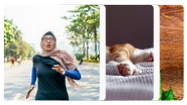
Lists

Staff Picks

566 stories · 673 saves

Stories to Help You Level-Up at Work

19 stories · 436 saves



Self-Improvement 101

20 stories · 1256 saves

Productivity 101

20 stories · 1153 saves

Talha Şahin

High-Level System Architecture of Booking.com

Hello everyone! In this article, we will take an in-depth look at the possible high-level...

8 min read · Jan 10

1.7K 13

MrDevSecOps

Introduction to SonarQube and Its Features

What is SonarQube?

5 min read · Aug 18, 2023

6

Benoit Ruiz in Better Programming

Advice From a Software Engineer With 8 Years of Experience

Vaishnav Manoj in DataX Journal

JSON is incredibly slow: Here's What's Faster!

Practical tips for those who want to advance in their careers

22 min read · Mar 20, 2023

13.7K 256

Unlocking the Need for Speed: Optimizing JSON Performance for Lightning-Fast Apps...

16 min read · Sep 28, 2023

12.4K 145

See more recommendations