Native Web Audio API Plugins

Jari Kleimola
webaudiomodules.org
jari.kleimola@gmail.com

Owen Campbell
Amptrack Technologies AB
owen@ampedstudio.com

ABSTRACT

This work enables native audio plugin development using the Web Audio API and other web technologies. Hybrid forms where DSP algorithms are implemented in both JavaScript and native C++, and distributed forms where web technologies are used only for the user interface, are also supported. Various implementation options are explored, and the most promising option is implemented and evaluated. We found that the solution is able to operate at 128 sample buffer sizes, and that the performance of the Web Audio API audio graph is not compromised. The proof-of-concept solution also maintains compatibility with existing Web Audio API implementations. The average MIDI latency was 24 ms, which is high when comparing with fully native plugin solutions. Backwards compatibility also reduces usability when working with multiple plugin instances. We conclude that the second iteration needs to break backwards compatibility in order to overcome the MIDI latency and multi-plugin support issues.

1. INTRODUCTION

The first Web Audio API implementation appeared in Google Chrome v10 (released 2011), followed by Safari (2012), Firefox and Opera (2013), and Edge (2015). The API has matured ever since its first introduction, and the specification is now transitioning to final standardization stages [1]. The specification exposes 21 AudioNodes, which are modular building blocks that – when connected together into an audio graph – implement composite DSP algorithms such as audio effect processors, synthesizers and other audio related web applications. These high level DSP algorithms are patched together using JavaScript. A recent addition called AudioWorklet [2] also enables custom, low-level DSP algorithm development to extend the available set of building blocks.

AudioWorklets run JavaScript (JS) and WebAssembly (WASM) code. The latter enables the use of general purpose programming languages such as C++ in web audio development [3]. Domain specific languages such as Faust [4] and Csound [5] are also supported through WASM. Though C++, Faust and Csound are commonly used in native audio plugin development, pure JavaScript and WASM implementations have not yet been viable in desktop Digital Audio Workstations (DAWs). This work explores various options to achieve that goal, and provides a proof of concept solution that enables any Web Audio API code to run as a native desktop audio plugin (see Figure 1).

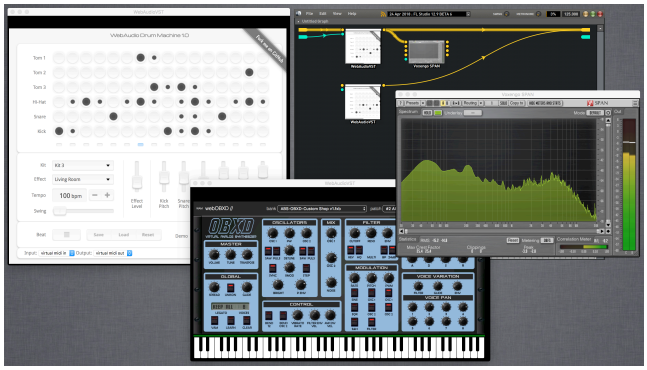

Figure 1. Native plugin host (top right) with two native Web Audio API plugins (left), and a VST spectrum analyzer.

The concrete contributions of the present work are: i) a list of mandatory requirements for bridging the gap between native plugin and web environments, ii) a survey of implementation options, iii) proof-of-concept implementation based on the most promising option, and iv) three native Web Audio API plugins for its evaluation. Source code for the implementations is available at **https://github.com/jariseon/web-audio-vst**

The organization of this paper is as follows. Section 2 details the requirements and explores options for their implementation. The proof-of-concept solution is described in Section 3, and evaluated in Section 4. Section 5 concludes.

2. REQUIREMENTS AND OPTIONS

2.1 Requirements

A native audio plugin is a shared code library which a plugin host application loads dynamically during runtime. The functionality of an audio plugin is divided into two parts. The *controller* handles non-real-time tasks such as configuration, plugin lifecycle management, preset loading and saving, and so on. It also usually provides a graphical user interface (GUI) for plugin parameter manipulation. The *processor* performs block-based audio processing and rendering tasks in real-time, and routes MIDI and automation events to the plugin's DSP engine. Because of real-time performance requirements, hosts run these two parts in separate threads.

The most prominent requirements for a native Web Audio API plugin are as follows. The plugin must:

Web Audio Conference WAC-2018, September 19–21, 2018, Berlin, Germany.

R1 provide an environment to run Web Audio API
R2 operate within a shared dynamic link library
R3 provide a mechanism for native ↔ browser communication
R4 be able to host a GUI
R5 process audio IO streams in real-time
R6 provide a mechanism for time-stamped MIDI IO

This work will focus on these requirements in the context of the VST2 [6] plugin standard, which despite its age and shortcomings is still the most ubiquitous cross-platform native audio plugin format. Our goal is to maintain maximum compatibility with existing Web Audio API implementations.

2.2 Implementation Options

R1 - *Web Audio environments*

This requirement calls for a JS engine and implementation of the Web Audio API. In addition to browsers themselves, native webview widgets encapsulate both operating system specific JS engines (Nitro/MacOS, Chakra/Windows, JSC/Linux), and at least partial Web Audio API implementations. Most well-known third party application frameworks offer webviews as well.

Web technologies have become increasingly popular in native application development, thanks to frameworks such as NW.js[1], Electron[2] and Chromium Embedded Framework (CEF) [7]. Their main architectural difference is that CEF is able to operate either as a standalone or as a webview, whereas NW.js and Electron run only as standalone applications. All three are based on Chromium content module, which offers the core functionality of Google's Chrome browser including its V8 JS engine and Web Audio API implementation. NW.js and Electron also embed Node.js[3], which integrates the JS engine with custom script and binary code modules. Partial Web Audio API implementations[4,5] are available as such Node.js modules.

In summary, environments for running Web Audio API include browsers, webviews, and Chromium-based frameworks (see Figure 2). The latter environments, including Google Chrome browser, currently offer the most up-to-date Web Audio API implementation. On the other hand, native webviews are the most lightweight solutions, as they are built into the operating system.

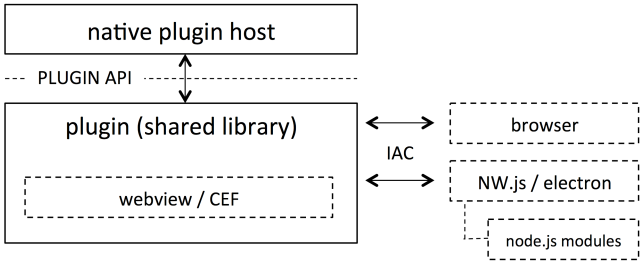

Figure 2. Environment options illustrated with dashed boxes.

R2 - *operating as a shared library*

Webviews (including CEF) are able to operate within shared libraries, whereas browsers and NW.js/Electron require inter-app communication (IAC) mechanism between the shared plugin library and the remoted Web Audio API application. The web audio environment thus needs to either run in-process inside the native plugin, or as a remote service through an IAC mechanism (e.g., shared memory or (web)sockets). In-process is the preferred option, as it minimizes latency and reduces the amount of architectural complexity.

R3 - *link between native and browser contexts*

Webviews (including CEF) provide inter-process communication (IPC) APIs for linking the native plugin environment with the browser context running Web Audio. These APIs allow asynchronous calls from native side to browser side, and vice versa. Most APIs only support textual data, and therefore binary data needs to be transformed into the less efficient base64 format. An alternative approach is to use websockets, which enable both textual and binary transfers. In that scenario the native plugin implements a websocket server that accepts connections from a localhost webview or remoted application. The downside of websocket implementations is increased complexity and potential conflicts with websocket port number allocations.

R4 - *GUI embedding*

Web technologies are well suited for plugin GUI development, and provide an attractive platform even for traditional audio plugins that implement their DSP in native code. In native plugin architectures the host provides a top level window, into which the plugin is expected to embed its GUI as a subview. Webviews (including CEF) support this requirement, whereas standalone application GUIs are in separate windows external to the host's window. Though both paradigms support this requirement, the distinctions are notable from a user experience perspective.

Operating system webviews are by definition not cross-platform solutions. In contrast, CEF provides a uniform user experience across platforms. The CEF release distribution format footprint is 50-100MB (depending on included functionality), which makes its separate bundling with each individual plugin installation impractical. We found however that with proper settings separate bundling is unnecessary, as will be discussed later.

R5 - *real-time audio IO*

Web audio runs in a high priority thread which is not directly accessible from the browser's JS main thread. The deprecated ScriptProcessorNode (SPN) is able to push and pull audio buffers to/from Web Audio API graph, although with a cost of increased latency: the minimum SPN buffer size is 256 samples, and due to thread-hopping and double-buffering, the total added latency is 512 samples. SPN is also susceptible to audio dropouts when the main application thread is busy with other duties. The recently released AudioWorkletNode/Processor pair cuts down the total latency to 256 samples, but is still prone to audio glitches due to garbage collection operations. IAC requires another double-buffering round, increasing total latencies to 1024 and 512 samples, respectively. This is unacceptable in pro audio environments.

Another option is to use a MediaStreamAudioDestinationNode, and couple it with a local WebRTC peer implementation in the native plugin, or in the remoted app. However, WebRTC audio streams are lossy and come with an increased latency. Opus codec running in CELT mode provides the best option in terms of sound quality and latency, but this is still suboptimal for pro audio scenarios.

[1] https://nwjs.io
[2] https://electronjs.org
[3] https://nodejs.org
[4] https://github.com/LabSound/LabSound
[5] https://github.com/audiojs/web-audio-api

The most straightforward way to hook into web audio stream is via a third-party virtual audio cable[6]. Unfortunately current browser engines provide only limited access to audio device output selection. Virtual audio cable routing within plugin host device chains is also complicated (if not impossible), producing detrimental effects to creative workflows. In addition, low level device driver installation-calls for administrator privileges.

Finally, environments based on the Chromium content module provide an alternative in-process hook into the web audio graph. This requires patching and recompiling a custom Chromium version, but provides a preferred solution in terms of latency and integration.

R6 - *MIDI*

Web Midi is supported in Chrome browser and Chromium based frameworks. Other environments can use IPC APIs to pass MIDI between JS and native plugin using interop or websockets. Web Midi requires user permission, which is unfortunately unsupported in current CEF version. This requires another Chromium patch, but would again provide an in-process solution to the problem.

2.3 Summary

Table 1 summarizes the implementation options for available environments (rows) versus requirements (columns). The cells marked with in-process and interop indicate preferred options in terms of seamless integration, efficiency and minimal latency. Note however that the cells enclosed in parentheses in column R5 offer only suboptimal audio IO routing because of inefficient hooks based on SPN, AudioWorklet or WebRTC.

As can be seen, CEF provides the most optimal support. The lack of direct audio and midi stream access make standard webviews less attractive. Browsers and Chromium-based application frameworks are standalone applications, and therefore not well suited for the task at hand. The remaining sections thus propose a solution based on CEF and patched Chromium. Our design goal is to minimize the the amount of required modifications in Chromium codebase, while still satisfying the requirements listed in section 2.1.

Table 1. Implementation Options

R1	R2	R3	R4	R5	R6
browser	n/a	iac	external	(iac)	midi/iac
nwjs/electr	n/a	iac	external	(iac)	midi/iac
webview	in-process	interop	in-process	(interop)	interop
cef	in-process	interop	in-process	in-process	interop

3. IMPLEMENTATION

The implementation process was iterative, but consisted of the following major stages. A VST2 skeleton plugin was first developed in Xcode 9.3. The CEF source code distribution 3.3359 (including Chromium stable 66) was downloaded, patched and compiled. The proof-of-concept plugins were developed as the final step.

3.1 System Architecture

The system architecture is shown in Figure 3. The native plugin host first loads and instantiates the WebAudioVST plugin, and enters a communication phase using VST API calls. The plugin creates a CEFBridge instance, which in turn loads and initializes the CEF dynamic library. CEFBridge hides the implementation details of the webview, and communicates with CEF through the CEF API. Chromium, and consequently CEF, operate in a multi-process configuration. The browser process interfaces the operating system, and launches the renderer process(es). The renderers are sandboxed and among other things, responsible for JavaScript execution. The browser and renderer communicate using asynchronous messaging. Chromium's audio streams are however synchronous thanks to a paired socket / shared memory mechanism between the processes.

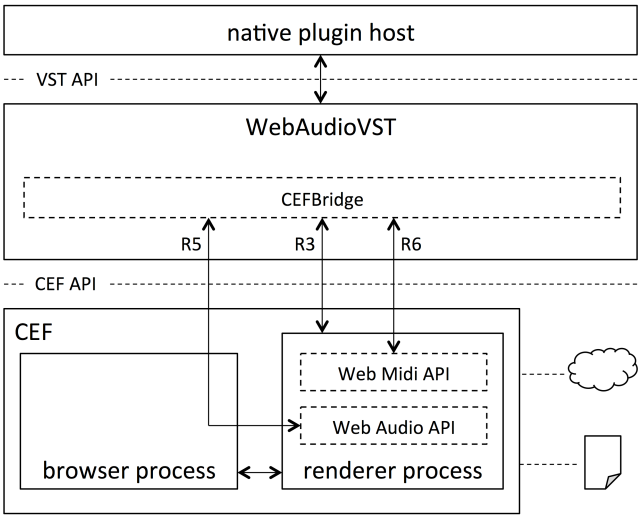

Figure 3. System architecture.

3.2 R2 - Operating as a shared library

Before proceeding with other requirements, we first ensured that CEF operates within a VST2 plugin as expected, and that it does not require modifications to the VST host. This proved to be the case with the following caveats. The CEF message loop needs to be integrated with the plugin host using the non-blocking `DoMessageLoopWork()` function. This function needs to be called repeatedly to perform a single CEF message loop iteration (see `idle()` function in Section 3.4). We also found that CEF initialization is best performed during the first `idle()` call: plugin main window is only partially realized in prior VST callbacks.

3.3 R3 – Controller API

We want to emphasize that the proposed solution is capable of running arbitrary Web Audio API implementations inside native Digital Audio Workstations (DAWs). However, to get the greatest benefit from the solution, we are introducing a controller API to streamline native Web Audio API plugin development.

The controller API provides an asynchronous communication channel between the native plugin C++ environment and the JS context in the embedded webview. CEF provides mechanisms for calling JS functions from the native side, calls from JS to the native side, as well as binding to browser events at the native side. The CEFBridge class hides the sandboxing-induced complexity with two RESTful methods, which are implemented at both JS and in native code. `postMessage(data)` sends arbitrary text format data from one endpoint to the other. The message eventually appears at the other endpoint's `onmessage(data)` event handler. This is similar to the Web Audio API AudioWorklet

[6] https://github.com/mattingalls/Soundflower, http://jackaudio.org

implementation, and in that sense one embodiment of this work (i.e., the hybrid plugin form) can be viewed as an external AudioWorkletProcessor that is implemented in native C++ code. The data passed between the endpoints is formatted as JSON, and is compliant to the Web Audio Plugin payload proposal [8, section 4.2]. A composite JS Web Audio Plugin class will thereby interface the native WebAudioVST plugin without additional wrappers.

The WebAudioVST plugin is initially configured as a stereo in/out processor. A Web Audio implementation conforming to the controller API publishes its descriptor by calling `postMessage ({verb:"set", key:"descriptor", value:descriptor})`. The descriptor JSON contains information about the number of audio IO ports and their channels, midi IO setup, preferred window size, parameter space and so on. CEFBridge parses the descriptor and informs the host with a VST2 API `ioChanged()` reconfiguration request. Web Audio implementations that do not conform to the controller API fall back to the default stereo in/out configuration.

3.4 R4 - GUI

VST2 supports generic and custom GUIs. The host creates generic GUIs by iteratively querying the plugin parameter space using indexed `getParameter*()` calls to retrieve the name, display string, unit and current value of each parameter. The CEFBridge class transforms the received descriptor (if any) into VST2 compliant responses. Custom GUIs are extended from the VST2 `AEffEditor` interface class. In this case, the host provides a top level window and calls the `open()` method in the GUI interface class. The plugin may then embed the CEF webview as a subview of the provided window. We found that the top level window is not yet instantiated in the operating system window manager at the time of the `open()` call, and therefore embed fails. The issue was solved by delaying CEF webview instantiation until the first `idle()` function call. The host calls this function repeatedly during runtime, which provides a means for CEF messageloop integration introduced in Section 3.2.

The proposed solution also supports use cases where the embedded webview only implements the GUI, while audio processing is performed in native C++ code running in the plugin. During the *prototyping* stage, plugin developers may use an external localhost HTTP server to provide HTML/CSS/JS content that implement the browser GUI. This also enables live source code editing and debugging while the plugin is running natively in the host. During runtime, the GUI (running in the webview) and the plugin (running natively inside the host) communicate using the asynchronous controller API mechanism described in section 3.3.

During the *deployment* stage, HTML/CSS/JS files may be packaged into a single zip file which is distributed with the plugin. The zip file may be protected with a password and encrypted for elementary intellectual property protection. The CEFBridge initialization phase first loads and decompresses the (encrypted and) zipped content using CEF functions. The bridge then navigates to the index.html file, and exploits CEF resource request interception to serve the files for the GUI implementation. However, zip encapsulation is beyond the proof-of-concept implementation scope, and is included here for completeness of discussion.

We found that plugins are able to share a single CEF installation instance, and therefore it needs to be downloaded and installed only once. This is achieved with matched settings in plugin and CEF bundles, as detailed in the source code release associated with this paper.

3.5 R5 - Audio IO

VST2 block-based DSP processing takes place in the plugin host's audio thread. The host pushes and pulls audio buffers to/from the plugin by repeatedly calling the `processReplacing` method with three arguments. The first two are arrays of input and output audio buffer pointers, while the third one determines the number of samples the host is expecting to push/pull. The CEFBridge class simply delegates the processing call to the custom Chromium patch implemented in this work.

The plugin and CEF browser process (see Figure 3) are in the same address space. The VST2 `processReplacing()` call propagates through standard CEF DLL bindings, and ends up in the browser process. The custom Chromium patch replaces Chromium's existing native AudioManager functionality - which is responsible for interfacing native audio IO by pushing/pulling web audio to/from the renderer process - with a cross-platform `VirtualAudioManager` implementation that pushes and pulls the audio buffers from/to VST2 plugin instead.

The actual streaming is provided by `VirtualAudioInputStream` and `VirtualAudioOutputStream` classes, also produced in this work. These classes wrap VST2 audio buffers in AudioBus structures, and call Chromium's `AudioInputCallback::OnData` (to push VST2 audio into the Web Audio API graph), and `AudioSourceCallback::OnMoreData` (to pull from the Web Audio API graph). Each Web Audio API AudioContext opens a dedicated output stream.

The custom implementation passes an additional fourth parameter to Chromium's browser process. The parameter holds the VST host's current time, which is passed along in `OnData` and `OnMoreData` calls. The time appears in the AudioContext `currentTime` attribute, and facilitates sample accurate Web Audio API rendering in sync with VST host time. `VirtualAudioInputStream` (i.e., VST2 plugin input) is available as a `MediaStreamAudioSource`, and `VirtualAudioOutputStream` (i.e., VST2 plugin output) becomes `AudioContext.destination`.

Audio can naturally be pre/post-processed in the native plugin before or after pulling the Web Audio API graph. This enables hybrid Web Audio API plugins which are developed partly in JS and partly in C++.

3.6 R6 - MIDI

VST2 provides an array of timestamped MIDI events in a companion `processEvents` callback, which is invoked on the audio thread when there are events to be processed during the next `processReplacing` audio slice. Our aim here is to provide a virtual MIDI cable between the native plugin and Chromium's JS context.

We first patched the Chromium renderer process to grant permission to `navigator.requestMIDIAccess`. Another patch was prepared for Chromium's browser process `MidiManagerMac` class, to create a single virtual midi input/output port pair instead of native MIDI endpoints collected from CoreMidi. This approach works well for a single plugin instance. The pool of available Chromium MIDI ports is however shared with all JavaScript contexts, which is undesirable for multi-plugin scenarios. This calls for an alternative approach that restricts each JS context to access only its specific virtual MIDI ports.

We ended up patching `navigator.requestMIDIAccess` and other Web Midi classes in JavaScript. The patch creates a virtual MIDI port pair, and links them with native side C++ endpoints provided by CEFBridge. The C++ endpoints reside in the CEF renderer process (see Figure 3), which in turn is connected to the CEF browser process via IPC layer. The CEF browser lives in plugin address space to interface with VST2 `processEvents` and `sendVstEventsToHost` calls. The patch is injected from the local file system into the JS context prior to content loading, and does not require modifications to existing scripts loaded from web.

An additional benefit of the proposed solution is that, unlike in standard browser implementations, MIDI message timestamps are in sync with the `AudioContext.currentTime` (and therefore plugin host song time). The timestamps are also sample accurate. This is achieved by reflecting current song position and the `deltaFrames` field of a `VstEvent` in the MIDI message timestamp.

4. EVALUATION AND DISCUSSION

Three proof-of-concept native Web Audio API plugins were implemented for evaluation. The first plugin is a minimal JavaScript DSP effect that re-implements VST2 AGain starter project using A) the stock GainNode and B) an equivalent implementation as an AudioWorklet. Its Web Audio API graph is MediaStreamSource → Gain/AudioWorklet → DestinationNode. The GUI is implemented in HTML/CSS and equipped with a toggle to switch between GainNode and AudioWorklet, and a single knob for gain control (see Figure 4). The plugin HTML file is loaded from the local file system, and it complies with the controller API of Section 3.3. The second plugin allows navigation to any URL in order to evaluate compatibility with existing Web Audio applications. The third plugin is a hybrid JS/C++ solution for latency evaluation: the plugin responds to MIDI note on/off messages and renders a Web Audio API square wave oscillator, positive impulse train from an AudioWorklet, and negative impulse train from native C++. Impulses are rendered at the start of each render buffer while MIDI note is active.

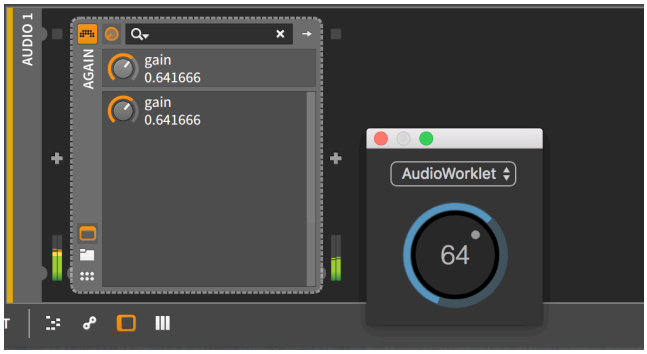

Figure 4. AGain native Web Audio API plugin in Bitwig.

The plugins were evaluated in two host applications: *FL MiniHost Modular*[7] v1.5.7 is able to load multiple plugins into a native equivalent of the Web Audio API graph, where each plugin is a node (see Figure 1). *Bitwig*[8] v2.3.4 (demo version) is a Digital Audio Workstation, where plugins are cascaded into device chains as shown in Figure 4. Evaluation was performed using MacBook Pro laptop Mid-2014, 2.2 GHz i7, 16 GB RAM, running MacOS 10.13.5.

4.1 Evaluation

Plugin host buffer sizes were set to 128 samples, which is the render quantum of Chromium's Web Audio API implementation. All plugins were able to produce glitch-free audio with the 128 sample (3ms) buffer size, without extra FIFO buffers in the rendering pipeline. However, we noticed occasional artefacts with the second plugin when navigating to URLs with relatively complex audio graphs. This was expected, since Chromium's native MacOS AudioManager doubles the 128 sample buffer size when pulling the graph. WebAssembly AudioWorklets operated without issues and with a good performance.

CPU load was evaluated using MacOS Activity Monitor. The load in the renderer process (Helper app) was dependent on the complexity of the audio graph and the amount of visual rendering. The renderer process CPU load was identical to the load inspected from Chrome, when navigating to the corresponding URL with the browser.

FL MiniHost called `idle()` method periodically at 54 Hz refresh rate on average. Updates were however interrupted while interacting with the MiniHost main window, albeit audio remained free of dropouts even if plugin screen turned static. Bitwig's plugin screen refresh rate was initially similar, but after a certain time period `idle()` calls were interrupted permanently. More robust operation was achieved by running CEF message loop integration in a separate thread. This also enabled independent screen refresh rate control. A related CEF limitation however disables GPU acceleration when the CEF message loop is integrated into host application (which is mandatory for this work). We hope that future CEF versions remove this limitation.

Latency was evaluated by mixing together three oscillators (see Figure 5): stereo Web Audio API square wave, positive impulse train AudioWorklet (right channel, bottom) and negative impulse train from native plugin (left channel, top). All sources responded to the same MIDI note on/off event.

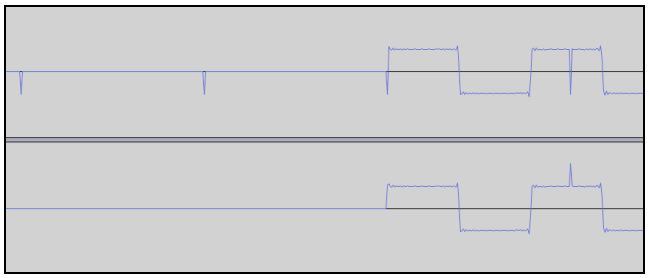

Figure 5. Latency (see text). Onsets are 128 samples apart.

As can be seen, the native plugin impulse train starts before the Web Audio API square wave (two render quanta in the figure), and AudioWorklet positive impulse train (at bottom) follows exactly one render quantum after the square wave onset. Onset deviations are due to asynchronous MIDI messaging: native plugin reacts to the events immediately, but when forwarded to the webview, the events have to first enter the renderer process main thread, and then travel over to the audio thread in AudioWorkletGlobalScope. We found that event forwarding was more performant with CEF SendProcessMessage (SPM) when coupled with a bound JS function at renderer side. With SPM the MIDI latency was 9 render quanta on average (median 8.5, min 3,

[7] https://www.image-line.com/support/FLHelp/html/plugins/Minihost Modular.htm
[8] https://www.bitwig.com

max 15). Direct CEF ExecuteJavaScript (EJS) function averaged 12.2 and (13.5, 4, 16), respectively. Variance was high in both cases.

Offline processing (i.e., freeze or bounce track in the host DAW) failed to work. This is a challenging task for any system that operates with asynchronous events and complex thread scheduling. Bitwig's real-time bounce however worked perfectly.

The HTML/CSS/JS files for the first plugin were loaded from the local filesystem. This worked without issues. The first plugin also complied to the controller API, which also worked well (see Figure 4). We noticed however that hosts do not respond to VST2 `ioConfig()` request properly. This is problematic for initialization and requires descriptor duplication in native code.

CEF binaries were not bundled in the plugins. Therefore it seems to be possible to share a single CEF installation between different plugins. The number of input MediaStreams is limited to one instance, even if opened from separate browser/renderer processes. This is in contrast to output streams, where each AudioContext opens a new stream.

4.2 Discussion

The performance of native Web Audio API plugins exceeded our expectations, and the proposed solution holds a definite promise for their implementation. Security is naturally compromised when comparing with day-to-day internet browsing, but we do not see that a prominent an issue: the proposed solution still runs Web Audio API in a sandbox, while traditional native audio plugins have full access to host machine. The evaluation however brought up issues which need to be addressed in a forthcoming iteration.

The most prominent issue is MIDI latency. Since one render quantum is 128 samples, SPM function average latency at 48 kHz is 24 ms. In addition, high variance in latency times makes host's latency compensation efforts impossible. Future iteration needs to pass MIDI events with audio buffers instead of sending them as asynchronous messages. This enables sample accurate rendering and solid latency compensation for AudioWorklets.

The second issue is multi-plugin support: due to limited input stream access the current solution can only support one Web Audio API effect plugin instance. And while it is possible to run several output streams in parallel (e.g., multiple synthesizers or drum machines do not pose an issue), the association between output stream and the plugin instance is not straight forward. With these in mind, it seems that instead of patching input and output audio endpoints in the CEF browser process, it might be beneficial to introduce a new pair of Web Audio API nodes specifically tailored for the scenario at hand. These nodes may also serve as generalized `ExternalAudioSource/Destination` nodes to support other use cases. Introducing new nodes would however break compatibility with existing Web Audio applications, which was one of the goals of this work.

The third issue is related to VST2 specification, which is not anymore up-to-date with modern plugin requirements. VST2 standard is informal, and leaves room for interpretation. Hosts do not unfortunately behave alike, which complicates adoption. We are planning to address this by using a plugin framework such as iPlug2 [9] or JUCE, which have tackled the problem already.

5. CONCLUSION

This work enables rapid native audio plugin prototyping and development using the Web Audio API and other web technologies. Native plugin audio input is captured using a Web Audio API `MediaStreamAudioSource` node, while native plugin audio output is available as `AudioContext.destination`. Native host MIDI is routed through virtual Web Midi ports.

Requirements were first collected and strategies for their implementation explored. The most promising solution was found to be based on Chromium Embedded Framework (CEF), which was then modified to suit the requirements. Evaluation proved that the solution maintains compatibility with existing Web Audio API implementations, albeit at a cost of increased MIDI latency and reduced usability when running multiple plugins simultaneously.

Future work comprises A) dropping backwards compatibility to improve MIDI latency and multi-plugin support, B) expanding from VST2 into other plugin formats by embedding CEFBridge in plugin frameworks like iPlug2 and JUCE, and C) ensuring compliance with most popular plugin hosts and a proposed Web Audio Plugin standard [10]. We are also considering binary distribution by committing the implemented Chromium patch to be integrated into future CEF releases.

6. REFERENCES

[1] Adenot, P., Toy, R., Wilson, C., and Rogers, C. 2018. *Web Audio API*. W3C Working Draft, June 19, 2018. Available online at http://www.w3.org/TR/webaudio/. (editor's draft at http://webaudio.github.io/web-audio-api/).

[2] Choi, H. 2018. AudioWorklet: The future of web audio. In *Proc. 43rd Int. Computer Music Conference* (ICMC-2018), Daegu, Korea.

[3] Kleimola, J. and Larkin, O. 2015. Web Audio modules. In *Proc. 12th Sound and Music Computing Conference* (SMC 2015), Maynooth, Ireland.

[4] Letz, S., Orlarey, Y., and Fober, D. 2017. Compiling Faust Audio DSP Code to WebAssembly, In *Proc. 3rd Web Audio Conference* (WAC 2017), London, UK.

[5] Lazzarini, V., Costello, E., Yi, S., and Fitch, J. 2014. Csound on the Web. In *Proc. Linux Audio Developers' Conference* (LAC-2014).

[6] Steinberg, VST3 : New Standard for Virtual Studio Technology,https://www.steinberg.net/en/company/technologies/vst3.html

[7] Chromium Embedded Framework, https://bitbucket.org/chromiumembedded/cef

[8] Buffa, M., Lebrun, J., Kleimola, J., Larkin, O., and Letz, S. 2018. Towards an open Web Audio plug-in standard. In *Companion Proc. The Web Conference 2018* (WWW '18). Lyon, France.

[9] Larkin, O., Harker, A., Kleimola J. 2018. iPlug 2: Desktop Audio Plug-in Framework Meets Web Audio Modules. In *Proc. 4th Web Audio Conference* (WAC-2018), Berlin, Germany.

[10] Buffa, M., Lebrun, J., Kleimola J., Larkin O., Pellerin, G., and Letz, S. 2018. WAP: Ideas For a Web Audio Plug-in Standard. In *Proc. 4th Web Audio Conference* (WAC-2018), Berlin, Germany.