WAP: Ideas for a Web Audio Plug-in Standard

Michel Buffa, Jerome Lebrun
Université Côte d'Azur
CNRS, INRIA
(buffa, lebrun)@i3s.unice.fr

Jari Kleimola, Oliver Larkin
webaudiomodules.org
(jari, oli)@webaudiomodules.org

Guillaume Pellerin, Stéphane Letz
IRCAM, GRAME,
guillaume.pellerin@ircam.fr
letz@grame.fr,

ABSTRACT

Several native audio plug-in formats are popular today including Steinberg's VST, Apple's Audio Units, Avid's AAX and the Linux audio community's LV2. Although the APIs are different, all exist to achieve more or less the same thing - represent an instrument or audio effect and allow it to be loaded by a host application. In the Web Audio API such a high-level audio plug-in entity does not exist. With the emergence of web-based audio software such as digital audio workstations (DAWs), it is desirable to have a standard in order to make Web Audio instruments and effects interoperable. Since there are many ways of developing for Web Audio, such a standard should be flexible enough to support different approaches, including using a variety of programming languages. New functionality that is enabled by the web platform should be available to plug-ins written in different ways. To this end, several groups of developers came together to make their work compatible, and this paper presents the work achieved so far. This includes the development of a draft API specification, a small preliminary SDK, online plug-in validators and a set of examples written in JavaScript. These simple, proof of concept examples show how to discover plug-ins from repositories, how to instantiate a plug-in and how to connect plug-ins together. A more ambitious host has also been developed to validate the WAP standard: a virtual guitar "pedal board" that discovers plug-ins from multiple remote repositories, and allows the musician to chain pedals and control them via MIDI.

1 - INTRODUCTION

The Web Audio API includes a set of unit generators called AudioNodes for graph-based audio DSP algorithms. The standard AudioNodes allow for developing a range of web applications that require audio engines that go beyond simple playback. The recent addition of the AudioWorkletNode provides an efficient way to implement custom low-level processing, significantly increasing the possibilities of this technology. There are many different apps created with the web audio API that run independently, however there is no standard way to make them interoperable i.e. take a drum machine developed by X, load it into an application developed by Y and apply audio processing developed by Z. In the native audio world, these interchangeable units are called "audio plug-ins" and applications that can use them are known as "hosts" which are typically DAWs.

The authors of this paper come from different research groups that have all been developing their own solutions for implementing audio plug-in-like entities in the browser. This paper discusses our ideas for a unified "Web Audio plug-in" standard (WAP) and the infrastructure surrounding such a standard. Other researchers' initiatives exist, such as the Web Audio API extension framework (WAAX) [5] and the work by Jillings et Al. [1] who proposed an "intelligent" audio plug-in framework for WebAudio with a JavaScript API. Our proposal differs, in that it aims to bring together several approaches already utilized by our groups, allowing web audio plug-ins to be coded in JavaScript, in C++ (via WebAssembly) or using DSLs. We would like to be able to support all approaches with a unifying Web Audio Plug-in standard. This should be flexible enough to support multiple approaches to plug-in development and consider future possibilities such as use in progressive web apps (PWA) or in native environments.

One of the groups involved created FAUST, a domain specific language (DSL) for audio DSP, which supports targeting Web Audio [4][8]. Another group created Web Audio Modules (WAMs) - an API for developing web-based plug-ins using C++ and WebAssembly [2], another group have been creating a variety of WebAudio applications including a virtual pedal board plug-in host and virtual amp simulators [9].

At the time of writing there are relatively few commercial audio-first products based on the Web Audio API, in comparison to the wide range of desktop audio software. These include, for example two web-based DAWs[1], a hearing test app[2], and an online music notation package[3]. This is likely to change since the introduction of the AudioWorklet, which will facilitate many more pro-audio use cases. Based on the shared interests of all the authors involved, and our observations about changes in web-based audio software, we believe there is a clear need for a high level audio processing/generating unit, as part of- or to work with the Web Audio API.

2 - CONTEXT

In a previous paper [6] we provided a state of the art of native, desktop audio plug-in formats, and described what makes the web platform different. The authors in [2] and [11] have also presented overviews of the defining characteristics of different native plug-in APIs. For the work in this paper we decided to go back and look in detail at the 2003 Generalized Music Plugin Interface (GMPI) final draft proposal[4], which, despite its age, provides a thorough overview of desirable qualities in an audio plug-in API. The LV2 plug-in API has been compared to the GMPI document in a categorized table that is published online[5]. We decided to make a similar comparison whilst making our Web Audio Plug-in specification[6] to guide our work. This enabled us both to identify the most important features a plug-in API should provide, but also to discard irrelevant specifications, and to think about the differences afforded by the web platform.

In the next sections we present the main features of our proposal as well as its current status.

[1] SoundTrap.com and BandLab.com
[2] healthy-hearing.mimi.io
[3] noteflight.com, ultimate-guitar.com
[4] https://tinyurl.com/k2wy5ge
[5] LV2 achievement of GMPI requirements: http://lv2plug.in/gmpi.html
[6] WebAudio plug-in vs LV2 vs GMPI: https://tinyurl.com/yd5fedrc

2.1 WAP and the GMPI

The GMPI draft proposal lists 114 requirements grouped into 23 categories. We divided the categories further into three groups based on their relevance to the fundamental requirements of an audio plug-in.

We decided that the first draft Web Audio plug-in specification should look at the following categories:

- **Host/plug-in Model**: We need to define how plug-ins are loaded, instantiated, and connected together. Hosting scenarios require mechanisms for plug-in discovery and host-plug-in interface description. We must also bear in mind that in web there might not be a dedicated host - a plug-in could run "standalone" in an embedded browser.
- **Events** and **MIDI**: there should be a way to send and receive events to / from plug-ins and host, and MIDI support is obvious for instruments.
- **Parameters**, **Persistence**: plug-ins will need to expose their parameter set and provide getter/setters
- **Plug-in Files**: More generally, a way to persist current state so that loading and saving of presets/banks can be implemented.
- **User Interfaces**: Although some plug-ins may prefer to run headless, we need to support both generic and custom GUIs.

The second category group will be targeted in a subsequent API version. The lower priority categories are: **Host Services, Time, Latency, Copy Protection, Localization, API Issues,** and **Wrappers**. Finally, some categories are irrelevant for Web environments, or already defined in the lower level APIs such as Web Audio and WebRTC. These include **Realtime Threading**, **Sample Rate**, **Audio I/O**, **Control I/O** and **Results**.
In addition, we need to take into account that GMPI is dated, and that several modern native plug-in APIs have been developed since 2003 [11]. The web browser environment also means that extra considerations must be added, that are not concerns for native plug-ins.
How a developer should write a plug-in or a host, what would be the main filename of a plug-in to load, its metadata, how the audio part and the GUI part should be handled, all these things should be specified and examples / SDK provided. For this, we will take inspiration from previous works done by the GMPI group and implemented by the LV2 standard.

2.2 Support for different WAP approaches

A Web Audio Plug-in standard should be able to support multiple approaches in terms of programming language and programming environment, including pure JavaScript, C++ (via WebAssembly) and domain specific languages. It should be possible to port existing code bases across to work as a WAP and DSLs should be usable for the audio processing part. For example the authors in [4] have developed the WAM API allowing the porting of native plug-ins to WAPs, and this has been demonstrated by porting several plug-ins originally made with JUCE [6]. The iPlug 2 framework supports the WAM format [11] and therefore could support WAPs, allowing existing iPlug plug-ins to be compiled to the format. The FAUST creators have developed a script to compile FAUST .dsp files to WAPs [4, 12], and more importers/exporters are on the way.

FAUST pedals, packaged as WAP plugins.

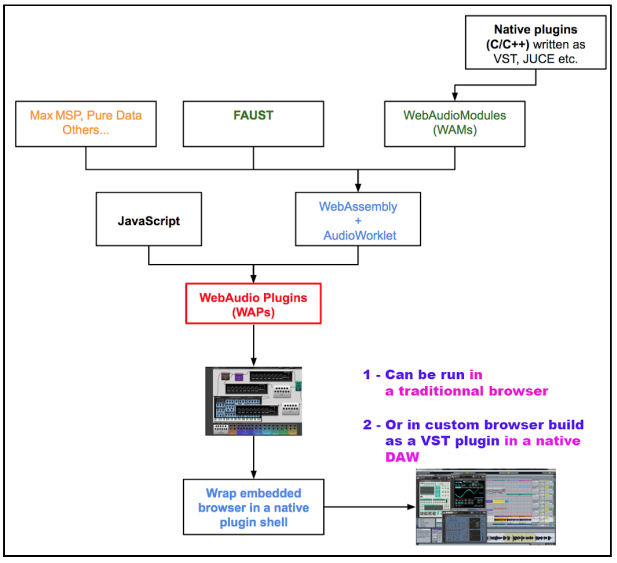

Figure 1: The WAP ecosystem. WAPs (in red) are at the center of both web and native worlds.

2.3 Support for Multiple Web Execution Environments

Although the web is the primary target for WAPs, there are use cases where integration with native applications may be required. For example, even if web DAWs offer very compelling features, experienced users may not be easily persuaded to switch to an online DAW immediately. In this case, supporting Web technologies in native apps could help making projects more portable across web and desktop platforms. Also the ability to test native plugins inside a familiar DAW before making a purchase, but without installation, might be an interesting use case. Native game engines and VR frameworks may also benefit from the Web Audio API which has good support for 3D audio. Options for bridging the web and native domains are explored in [10].

Fig. 1 shows WAPs as the pivot standard for all sorts of sources (JS, Faust, C++, etc.) and execution environments (standard browser, or browser embedded in a native plug-in, or as Progressive Web Apps or Chromium-based apps).

3 - CURRENT STATE OF OUR PROPOSAL

Our proposal consists of a draft specification, online tools and set of examples. Our driving vision is to build on top and integrate with existing web APIs, and to keep the proposed API

as minimal as possible.

3.1 A Draft Specification

The goal of our draft Web Audio plug-in API proposal [6] was to devise a minimal set of mechanisms that allow interoperation between our independently developed frameworks. A high level overview of the proposal is given below.

WAP extends AudioNode (or AudioWorkletNode) and thus inherits their familiar properties and methods. This ensures interoperation with standard Web Audio API nodes and applications built on top of web audio graph. Integration with Web Midi is provided by MIDIPort members.

WAPs are either *composite* or *custom* audio nodes. Composite nodes[7] encapsulates an audio subgraph that is built from any number of (elementary) AudioNodes. Custom nodes are AudioWorklets, with a fallback to a ScriptProcessorNode descendant. Although implementation details are outside the scope of the proposed API, a standard (but extensible) communication protocol between AudioWorkletNode and AudioWorkletProcessor was considered beneficial.

WAPs are GUI aware but agnostic about their implementation strategy. This means that WAPs may be headless, or they may expose a visual HTML element (e.g., div, canvas, svg, or custom element) which can be attached to DOM. *The WAP design will ensure that the GUI code is loaded only if necessary.*

WAP metadata describes implementation specific aspects of the plug-in. Metadata is available as a separate JSON file and also as a runtime object. Metadata describes audio and midi IO configuration, namespace attributes, parameter space, plug-in type, URIs and so on. WAP repositories may collect JSON files into aggregates for discovery purposes.

WAP endpoint is described by an URI, which may point to an online or local filesystem resource. Metadata may describe separate URIs for headless and GUI equipped WAPs. We foresee two embedding strategies: a hosting web page may simply employ one of the URIs in a script/link tag. More complex WAPs, such as those implemented in WASM may however require a dynamic loading mechanism.

3.2 Online Tools, Tutorials and Examples

Along with the online documentation[8], we propose simple examples/tutorials both for the "host side" and "plug-in side" of our proposal, as well as online tools such as validators/testers. Some are presented in the different figures that follow. Each legend contains a footnote with the link to the runnable webapp.

```
var ctx = new AudioContext();
var player = document.getElementById("soundSample");

var mediaSource = ctx.createMediaElementSource(player);
var pluginURL = "https://wasabi.i3s.unice.fr/WebAudioPluginBank/WASABI/PingPongDelay2";
var plugin = new WAPlugin.WasabiPingPongDelay(ctx, pluginURL);

plugin.load().then((node)=>{
  mediaSource.connect(node.getInput(0));
  node.connect(ctx.destination);
});
```

Figure 2: loading a headless plug-in from its URI, insert it in the WebAudio graph[9]

Host loading a headless plug-in: Fig. 2 shows extracts of a minimal host implementation that loads a headless plug-in and connect it to the WebAudio graph. Behind the scenes, a JSON metadata file is loaded from the plug-in URI. A `<script src="..."></script>` HTML tag is added if needed. Following that, the plug-in is initialized. Since it may load assets such as image files or a WASM module asynchronously, the `load` method returns a JavaScript promise. In this example, the name of the plug-in class is hardcoded but it could have been built dynamically from the content of the plug-in metadata JSON file (further examples shows how to do this). Let's notice that from a host point of view, the plug-in might be of any kind: a WebAudio graph in a CompositeNode or a single CustomNode (AudioWorklet) node, written in JavaScript or in WebAssembly, etc.

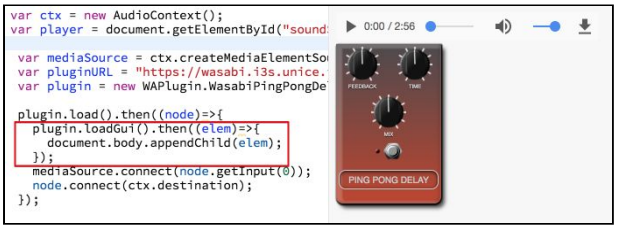

Figure 3: the same plug-in, with GUI. Note that the GUI code is downloaded in the host only when needed[10].

Host loading a plug-in with GUI: Fig. 3 shows the same example but this time, we also load asynchronously the GUI code (HTML, CSS, JS). The `loadGUI` method returns a single HTML element that contains the whole plug-in GUI. Here again, the method is asynchronous and returns a promise as a plug-in can have to load images for knobs, etc.

The `load` and `loadGUI` methods implementations are inherited by default when you extend the WebAudioPluginFactory class from the SDK, but can be overridden by the developer. In our examples, we use Web Components to package the GUI files in a single html file, adding encapsulation and avoiding any naming conflicts. Behind the scenes the default `loadGUI` method creates a `<link rel="import" href="main.html">` when needed. If one prefers to use a HTML canvas or Nexus UI for making the GUI, just override the `loadGUI` method.

[7] https://tinyurl.com/ybwm3bjy
[8] http://wasabihome.i3s.unice.fr/webaudio-plug-in-proposal/
[9] https://jsbin.com/bekenexefo/1/edit?js,output
[10] https://jsbin.com/xarotez/edit?js,output

Figure 4. Online plug-in tester[11]: enter the plug-in URI to validate the plug-in before publishing to a repository.

More detailed examples are available on the documentation pages of the WAP proposal. Some show in particular how to do real dynamic discovery, without hard coding any class names in the host code.

Plug-in online validator: this online tool uses this dynamic behavior and is provided to plug-in developers to test their work. Figure 4 shows an individual online plug-in tester. Copy and paste a plug-in URI and the code will be downloaded, the plug-in tested, and if a minimal set of tests passed, the plug-in will be runnable on the page and its GUI displayed, etc. You can then publish it on a repository. Notice that not all tests are mandatory to make the plug-in usable. For example, if a plug-in does not implement the load/save of its parameter state, it is still usable.

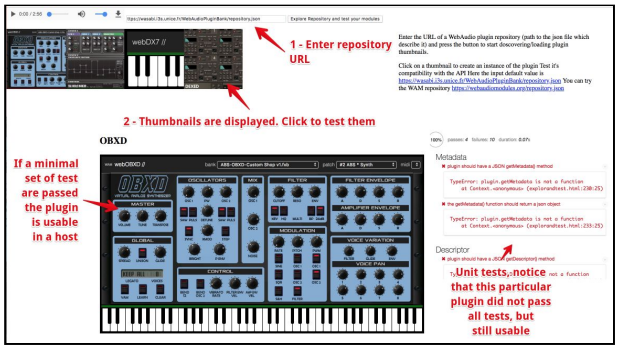

Figure 5: repo and plug-in tester[12].

Plug-in repository online validator: Fig. 5 shows a remote repository tester: enter the URI of a REST endpoint and the list of plug-ins (with associated URIs) is first fetched, and then, in a second time, each plug-in metadata file is also fetched. Each plug-in thumbnail is displayed on the page and can be clicked to test the corresponding plug-in. If mandatory tests passed, then you'll be able to try the plug-in online and get a full unit test report.

```
Promise.all([
    host.load("./../WASABI/QuadraFuzz/QuadraFuzz.html"),
    host.load("./../Faust/ZitaRev/FaustZitaRev.html"),
    host.load("./../Oliver-Larkin/ThruZeroFlanger/FaustThruZeroFlanger.html"),
    host.load("./../Oliver-Larkin/WeirdPhaser/FaustWeirdPhaser.html"),
    host.load("./../Oliver-Larkin/StereoFrequencyShifter/FaustStereoFreqShifter.html"),
    host.load("./../Oliver-Larkin/Blipper/FaustBlipper.html"),
    host.load("./../Oliver-Larkin/DualPitchShifter/FaustDualPitchShifter.html")
])
.then(function(nodes) {
    node[0].connect(node[1]);
    node[1].connect(node[2]);
    node[2].connect(node[3]);
    node[3].connect(node[4]);
    node[4].connect(node[5]);
    node[5].connect(node[6]);
    node[6].connect(ctx.destination);
});
```

Figure 6: loading and chaining multiple plug-ins[13]

A plug-in "loader" utility object: in some cases a developer might want to be sure that several plug-ins have been loaded before chaining them. The WAP SDK provides a "plug-in loader" utility object that can be used with the `Promise.all` method from ES6, as shown in Fig. 6.

Writing a plug-in, minimal steps: now that we looked at how a host can discover and use a plug-in, it is easier to illustrate the different steps necessary to make a plug-in. the SDK provides multiple classes that can be subclasses, utility classes and objects. When writing a plug-in, different parts should be considered:

Part 1 - The `main.json` metadata descriptor that will contain a set of key/values keys. Mandatory: plug-in name, vendor, version, thumbnail file (Fig. 7). Using the vendor and name values the main JS the class name of the plug-in and its relative URI can be inferred.

```
{
  "documentation": "BLAH BLAH",
  "name": "PingPongDelay",
  "thumbnail": "WasabiPingPongDelay.png",
  "vendor": "Wasabi",
  "category":"Delay",
  "version": "1.0"
}
```

Figure 7: minimal json file for describing a plug-in

Part 2 - The plug-in "main class" that will implement the WebAudio plug-in API. It should extends either the WebAudioCustomNode (i.e if the plug-in is an AudioWorklet) or the WebAudioPluginCompositeNode class (if it is made of a set of WebAudio nodes). We opted for a "convention over configuration approach"[14]: minimal steps are needed to have a runnable plug-in as many default values will be inherited. For example, the default main file will be `main.js` except if indicated in the metadata json file, `getParam/setParam` methods will be inherited, etc. Fig. 8 and 9 show source code extracts from the SDK classes that are provided for creating a CompositeNode plug-in, Fig. 10 shows a skeleton of what the main class of a plug-in would look like, and finally, Fig. 11 shows how this plug-in can be used as a regular WebAudio node. Full example is available online.

```
//----- 1 - CompositeAudioNode ----
// has connect/disconnect methods
// A custom composite node can be derived from this prototype.
class CompositeAudioNode {
  get _isCompositeAudioNode () {
    return true;
  }

  constructor (context, options) {
    this.context = context;
    this._input = this.context.createGain();
    this._output = this.context.createGain();
  }

  connect () {
    this._output.connect.apply(this._output, arguments);
  }

  disconnect () {
    this._output.disconnect.apply(this._output, arguments);
  }
}
```

[11] https://wasabi.i3s.unice.fr/WebAudioplug-inBank/testers/test2.html
[12] https://wasabi.i3s.unice.fr/WebAudioplug-inBank/testers/explorandtest.html
[13] https://tinyurl.com/ydg5yt32
[14] https://en.wikipedia.org/wiki/Convention_over_configuration

Figure 8: the `CompositeAudioNode` prototype from the WAP SDK

```
// ------------------------
// CREATE THE PLUGIN CLASS
// ------------------------
class WebAudioPluginCompositeNode extends CompositeAudioNode {
  constructor (context, options) {
    super(context, options);
    // ...
  }
  // P2 from WAP specification...
  set descriptor(descriptor) {
    this._descriptor = descriptor;
  }
  get descriptor() {
    return this._descriptor;
  }
  // ... other default properties and methods
  // that are indicated in the WAP spec
}
```

Figure 9: class from the SDK that implements parts of the WAP API (default values). Meant to be subclassed.

```
// API P1: a WebAudio plugin can be a CompositeNode or a CustomNode
class WasabiStereoDelay extends WebAudioPluginCompositeNode {

  constructor (context, options) {
    super(context, options);

    // P2 Plugin metadata
    // setter is inherited
    this.metadata ={
        "documentation": "BLAH BLAH",
        "name": "PingPongDelay",
        "thumbnail": "WasabiPingPongDelay.png",
        "vendor": "Wasabi",
        "category":"Delay",
    "version": "1.0"
    };

    // ... other settings params, descriptor etc...
    // Make the delay here
    this.buildAudioGraph();
  }
```

Figure 10: a composite plug-in should extend the `WebAudioPluginCompositeNode` class.

```
var context = new AudioContext();
var myDelayPluginCompNode = context.createWasabiStereoDelayCompositeNode();
var oscNode = context.createOscillator();

oscNode.connect(myDelayPluginCompNode).connect(context.destination);

oscNode.start();
oscNode.stop(1.0);
```

Figure 11:finally, a composite plug-in can be used like a regular WebAudio node[15] even if made of multiple nodes.

Part 3 - The plug-in "factory" class that will implement (or inherit) the `load` and `loadGUI` methods. Fig. 12 shows and example of such a factory method.

```
var WAPlugin = WAPlugin || {};

WAPlugin.WasabiStereoDelay = class WasabiStereoDelayFactory extends WebAudioPluginFactory {
    constructor(context, baseURI) {
        super(baseURI)
    }

    // load and loadGUI methods are inherited but
    // can be overriden here...
}
```

Figure 12: plug-in factory class. This is the class used by the host code to instantiate the plug-in.

3.3 The pedal board host application

Along with our "10-lines of code long example hosts", a "virtual pedal board" web app was developed [9] as a more ambitious host that gives the possibility to interactively chain plug-ins in order to create more complex sounds and configurations (Fig. 13). This pedal board host handles the discovery of plug-ins from multiple repositories (local or distant), global sound card I/O and gain adjustments, plug-ins' life cycles and interconnections, saving and restoring states/banks/presets.

Figure 13: plug-ins inside a virtual pedal board[16].

One can create instances of plug-ins by dragging and dropping their thumbnails into the main area. We can then position them, connect them together, etc. Finally, using their GUI (knobs, sliders, switches), we can adjust each plug-in individually. By assembling an amplifier simulator, a reverb, a fuzz and a stereo delay, for example, one is able to create a rich psychedelic sound. We did also contribute to the *webaudiocontrols library*[17] by adding MIDI support to all the GUI elements it offers (knobs, switches, etc.). Hence, the plug-ins receive the possibility of having their GUI controlled remotely via any MIDI controller. Fig. 13 shows a typical screen of this host application, with plug-ins written in FAUST, in C++ (WebAudioModules) and in JavaScript (guitar amp simulator, FX pedals). As explained in [10], this pedalboard has been also successfully run in custom browser builds packaged as VST plugins, bringing WAPs into native DAWs.

4 - FUTURE WORK

For the first draft of the WAP API, we isolated a minimum set of features so that implementations can follow quickly. Short terms improvements concern MIDI, plug-ins modularization and how Progressive Web Applications would impact the API and SDK if we want to make hosts and plugins available offline, or use the local file system, for example.

One of the main tasks for future work is to add full MIDI support for host-plugin synchronization and plug-in exchanges**.** In the current version, WAP plug-ins can receive midi events using the `onMidi` method from the WAP API, for instance to support midi-learn capabilities. However, in the future this API will need to be enhanced with virtual midi ports to support midi event processing pipelines. As global time and/or timecode shall be managed by the DAW (native or web oriented), MIDI MTC support will not be implemented in short term.

Offline webapps, hard disk use and relaxing constraints is possible: the mobile Web pushed to close the gap between native and web applications. For example, Progressive Web Applications[18] (PWA), based on standard W3C APIs such as the Service Worker API and the File and FileSystem APIs enable now web apps to run offline, and can give them

[15] https://jsbin.com/wadoqal/edit

[16] https://wasabi.i3s.unice.fr/dynamicPedalboard/

[17] https://github.com/g200kg/webaudio-controls, a WebAudio UI widget lib.

[18] https://blog.mozilla.org/firefox/progressive-web-apps-whats-big-deal/

privileges such as access to the hard disk when the user grants permission (e.g. by installing the app). Future relaxed constraints could include leveraging the priority of audio thread, or getting exclusive access to the sound card. We follow closely the evolution of this W3C initiative and will update WAPs accordingly.

Modularized plug-ins, support for multiple I/O: In order to be able to wrap anything into anything, as it is done in usual patching environments and modular synthesizer architectures, some plugins and components could be divided into several elementary sub-modules so that each one can be (re)patched differently. This could be achieved using MIDI signals for parameter control and modulation but also for audio signals implying that any number of internal signals can be derived from one plugin module and exposed to the plugin external I/O interface. Some nice scenarii could be then imaging coupling hardware and software, not only with MIDI streams but also with more generally continuous signals like CV/gate for example.

5 - CONCLUSION

We presented a proposal for an open WebAudio Plug-in standard (WAP), that consists in a draft specification, a small preliminary SDK, tutorial and examples written in JavaScript with online plug-in and repository validators. As of today, FAUST effects and instruments, as well as Web Audio Modules (A C++ plug-in API working with AudioWorklets/WebAssembly) are compatible with WAPs and more languages/environments will be supported. WAPs also use “composite nodes” to unify JavaScript plug-ins made of multiple WebAudio nodes and plug-ins that are single AudioWorklet nodes. We also developed a more ambitious host application that scans plug-ins repository (local or remote) and can be used to assemble multiple plug-ins in a visual graph (the pedal board web app). This app can be run in a traditional web browser but also as a native plugin in a custom browser build, bringing WAPs to reference native DAWs, making them “universal plugins”.

6 - ACKNOWLEDGMENTS

French Research National Agency (ANR) and the WASABI project team (contract ANR-16-CE23-0017-01).

7 - REFERENCES

[1] Jillings, N. and al. 2017. Intelligent audio plug-in framework for the Web Audio API. In *Proc. 3rd Web Audio Conference* (WAC 2017), London, UK.

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

[3] Buffa, M. & al. 2017. WASABI: a Two Million Song Database Project with Audio and Cultural Metadata plus WebAudio enhanced Client Applications. In *Proc. 3rd Web Audio Conference* (WAC 2017), London, UK.

[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] Choi, H. and Berger, J. 2013. WAAX: Web Audio API eXtension. In *Proc. Int. Conf. New Interfaces for Musical Expression* (NIME’13), Daejeon, Korea.

[6] 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. (April 23--27, 2018). 759-766. DOI= https://doi.org/10.1145/3184558.3188737

[7] Gallidabino, A. and Pautasso, C. 2018. The Liquid User Experience API. In *Companion Proc. The Web Conference 2018* (WWW '18). Lyon, France. (April 23--27, 2018). DOI=https://doi.org/10.1145/3184558.3188738

[8] LETZ, S., ORLAREY, Y., et FOBER, D. FAUST Domain Specific Audio DSP Language Compiled to WebAssembly. In : *Companion of the The Web Conference 2018 on The Web Conference* 2018.

[9] Buffa, M., Lebrun, J, WebAudio Virtual Tube Guitar Amps and Pedal Board Design, submitted to the *4th Web Audio Conference* (WAC 2018), Berlin, Germany.

[10] Kleimola J. and Campbell O., Native Web Audio API plugins, submitted to the *4th Web Audio Conference* (WAC-2018), Berlin, Germany.

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