Web Audio: Some Critical Considerations

Victor Lazzarini, Steven Yi and Joseph Timoney

[1]Sound and Music Research Group
Maynooth University
Maynooth, Co. Kildare Ireland

victor.lazzarini@nuim.ie, stevenyi@gmail.com

Abstract. *This paper reviews the current state of the Web Audio API, providing some critical considerations with regard to its structure and development. Following an introduction to the system, we consider it from three perspectives: the API design; its implementation; and the overall project directions. While there are some very good aspects to the API design, in particular its concise and straightforward nature, we point out some issues in terms of usage, Javascript integration, scheduling and extensibility. We examine the differences in browser implementation of builtin nodes via a case study involving oscillators. Some general considerations are made with regards the project direction, and in conclusion we offer a summary of suggestions for consideration and further discussion.*

1. Introduction

The Web Audio API[Adenot and Rodgers 2015] is a framework aimed at providing sound synthesis and processing as part of Javascript engines embedded in World-Wide Web (WWW) browser software. Such functionality had been previously only partially explored via plugin systems such as Adobe Flash. Since the introduction of the audio element in the HTML5 specification, basic streaming audio playback has been possible, but this has not been developed significantly to allow for more complex sound computing applications. These include capabilities provided by game engines, and by desktop audio software (such as mixing, processing, filtering, sample playback, etc.). The aim of the Web Audio API is to support a wide range of use cases, which is acknowledged to be a very ambitious proposition.

In this paper, we would like to raise a number of questions with regards to this framework, and explore some issues that have been left so far unresolved. The WebAudio API has seen some significant changes in the past two years, and is being strongly supported by the major browser vendors[Wyse and Subramanian 2013]. It has also been the main focus of a major international conference (the Web Audio Conference at Ircam, Paris [Ircam 2015]), where a number of projects employing this technology have been showcased (for a sample of software using the API, please refer to [Roberts et al. 2013], [Lazzarini et al. 2014], [Lazzarini et al. 2015], [Mann 2015], [Wyse 2015], [Monschke 2015], and [Kleimola 2015]). While these developments bring some very interesting possibilities to audio programming and to Ubiquitous Music, we feel it is important to consider a number of aspects that relate to them in a critical light.

Our interest in the Web Audio API is twofold: firstly, we hope it will eventually provide a stable environment for Music Programming, and add to the existing choice

of maturely-developed Free, Libre and Open-Source (FLOSS) sound and music computing systems (such as SuperCollider[McCartney 2015], Pure Data[Puckette 2015], and Csound [Ffitch et al. 2015]); secondly, we would like it to provide the supports we need to host efficiently a Javascript version of Csound [Lazzarini et al. 2015]. In the light of this, we would like to explore some of the issues that are currently preventing one or the other, or both, to come to fruition.

The paper poses questions that relate to a number of perspectives. From a technical side, we would like to discuss points of Application Programming Interface (API) design, and the split between builtin, natively-implemented, components (nodes) with Javascript interfaces, and the user-defined, pure-Javascript, elements which include the `ScriptProcessorNode` and the upcoming `AudioWorker`. We evaluate the current API according to requirements to meet various musical use cases, and see what use cases are best supported and what areas where the current API may present problems.

Complementing this analysis, we consider the issue where the Web Audio native components are implemented by the vendors in different ways, based on a specification that is open to varied interpretation. Since there is no reference implementation for any of these components, different ways of constructing the various unit generators can be found. As a case study, we will look at how the `OscillatorNode` is presented under two competing Javascript platforms, Blink/Webkit (Chrome) and Gecko (Firefox). We aim to demonstrate how these not only use different algorithms to implement the same specification, but also lead to different sounding results.

From a project development perspective, we have concerns that there is not a unified direction, or vision, for Web Audio as a system. Extensibility appears to be provided as an afterthought, rather than being an integral part of the system. This is exemplified by how the `ScriptProcessorNode` was provided to users with some significant limitations. These are due to be addressed with the appearance of the `AudioWorker`, whose principles are discussed in this paper. We also observe how the long history of computer music systems and languages can contribute to the development of the Web Audio framework.

2. The API and its design

The Web Audio API has been designed in a way that allows simple connections between audio processing objects , which are called `AudioNodes` or just *nodes* in this context. These connections are simply performed by a single method (`connect()`) that allows the output of one node to be put to another node. These objects all live within a `AudioContext`, which also provides the end point to the connections (physically, the system sound device), the `AudioContext.destination`. Aspects such as channel count are handled seamlessly by the system, and obey a number of basic rules in terms of stream merging or splitting. Such aspects of the API are well designed, and in general, we should commend the development team for the concise and straightforward nature of its specification.

In the API, the audio context is global: it controls the overall running of the nodes, having attributes such as the current time (from a real time clock), the final audio destination (as mentioned above), sample rate, and performance state (suspended, running, closed). This design can be contrasted with the approach in some music programming

systems such as Csound and SuperCollider, where local contexts are possible, on a per-instance/per-event basis. Allowing such flexibility can come with a cost of increased complexity in the API, but at the level at which the framework is targeted, it might be something that could be entertained.

In general, it is possible to equate Web Audio nodes with the typical unit generators (ugens) found in music programming systems. However there are some significant differences. One, which was pointed out in [Wyse and Subramanian 2013], is that there are two distinct types of nodes: those whose 'life cycle' are determined by start and stop commands, and those whose operation is not bound by these. This leads to a two-tier system of ugens, which is generally not found in other music programming systems. In these, the classification of ugens tends to be by the type of signal they generate, and in some cases by whether they are performing or non-performing (ie. whether they consume or produce output signals in a continuous stream). Such differences have implications for programming in that nodes that are 'always-on' can be more or less freely combined into larger components that can themselves be treated as new nodes, whereas the other type is not so amenable to this type of composition. This is not an optimal situation, as ideally, programmers should be able to treat all nodes similarly, and apply the same principles to all audio objects being used.

A related difficulty in the design is the absence of the concept of an instrument, which has been a very helpful element in other music programming systems. In these, they take various forms: patches (PD), synthDefs (SuperCollider), and instruments (Csound). They provide useful programming structures for encapsulating unit generators and their connecting graphs. In some senses, nodes that are activated/deactivated via start-stop methods implement some aspects of this concept, namely, the mechanisms of instantiation, state and performance. But in most other systems, instruments are programming constructs that are user-defined, encapsulating instances of the ugens that compose it. In other words, they sit at a different level in the system hierarchy. While we might be able to introduce the concept via a Javascript class, this is perhaps more cumbersome than it needs to be. The concept of an instrument could also allow the introduction of local contexts.

From another perspective, the Web Audio API does not offer much in terms of lower-level access to audio computation. For instance, users do not have access to the individual data output from nodes (outside the `ScriptProcessor` or `AudioWorker` nodes). It is not possible to control the audio computation at a sample or sample-block level, something that audio APIs in other languages tend to provide (e.g. PyO[Bélanger 2015] or the SndObj[Lazzarini 2008] library for Python). Such access would allow a better mix between natively-implemented nodes and Javascript ones.

2.1. ScriptProcessor and AudioWorker

The `ScriptProcessorNode` interface has been present in the API since the first published working draft (in the form of a `JavaScriptAudioNode`, as it was called then). The main aim of this component was to provide a means of processing sound through Javascript code, as opposed to the natively-compiled bultin nodes. This is currently the only means of accessing the individual samples of an audio signal provided by the API, but it sits awkwardly amongst the other built-in nodes, which are opaque.

More importantly, script processor code is run in the Javascript main thread, and asynchronously to the other nodes. It communicates with the rest of the audio context through `AudioBuffer` objects, and if these are not of sufficient size, dropouts may occur. Higher latencies are then experienced as the result of this. In addition, any interruption by, for instance, user interface events, can result in dropouts. These characteristics render the `ScriptProcessorNode` unsuitable for applications which require a robust system. They limit significantly the extendability of the system. Given that Web Audio is quite limited in terms of its offer of builtin nodes (if compared to other music programming systems), this represents a significant issue at the time of writing.

In order to rectify the problems with the script processor, a new node interface has been introduced in the latest Web Audio API editor's draft [Adenot and Rodgers 2015], the `AudioWorkerNode`. This follows the model defined for the Web Worker specification [Hickson 2014], which describes an API for spawning background threads to run in parallel with the main page code. The Audio Worker has two sides to it: the one represented by `AudioWorkerNode` methods, visible to the main thread; and another that is provided in the actual worker script that processes the audio. This is given by an `AudioWorkerGlobalScope` object, which allows access to the input and output audio buffers and other contextual elements. A script is passed to the Audio Worker on creation, and is run synchronously in the audio thread (rather than in the main thread as the script processor did). In the cases where the WebAudio implementation places this thread on high priority, using the Audio Worker will mean a demotion to normal priority, as for security reasons, Javascript user code is not allowed to run with higher than normal priority. Also, the specification dictates that the processing script cannot access the calling audio context directly. The key configuration parameter of the sampling rate is passed to the script as a readonly element of the `AudioWorkerGlobalScope` interface.

Since no actual implementation of the AudioWorkerNode exists at the time of writing, it is not possible to assess its performance. There are some indications that it might provide a more robust means of extending the Web Audio API, but some aspects of its design (such as the separation between the script context and the calling audio context) may limit it to some use cases. We understand this to be motivated by security reasons (as many of the design decisions in Javascript engine-provided APIs have to be), but inevitably it is a limitation of the current specification.

In providing Audio Workers, the editors of the Web Audio API are marking the `ScriptProcessor` node as deprecated. However, some applications for script processors might still be found, and so it could be advisable to keep providing this interface in future versions of the system.

2.2. AudioParams

AudioParams are exposed as parameters for AudioNodes. AudioParams can have a single value set, can be connected to from other nodes, or also automated with values over time. While the first two ways of setting values seem to align well with the rest of the API, the third option of automating values via function calls is somewhat of an outlier. Since automation times and values are set directly on the AudioParam itself, the curve values can not be shared with multiple params. Instead, if one wants to use the same automation values, one has to set the values for each parameter.

In systems such as Csound and SuperCollider, time-varying values using piecewise segment generators are often done using unit generators designed for that purpose. Within the context of WebAudio, a similar implementation could have been done by creating an AutomationNode. By using a node, the values of the automation could then be connected to multiple AudioParams. In that regards, the design of AudioParams adds another node-like source of values in the graph that is implicitly connected, rather than explicitly done so like other node inputs.

The user is certainly able to create and use their own automation nodes by implementing them in Javascript. This would also allow one to create other types of curves and means of triggering than those provided by the AudioParam API. However, since this appears to be a very basic functionality that could well be encapsulated as a node, it appears that it would be best handled by an addition to the API.

2.3. Scheduling

Scheduling issues are also worthy of note. In many similar systems, an event mechanism is provided or implemented behind the scenes. In Web Audio, there is no event data structure to schedule. Instead, as we have discussed above, the API encourages creating a graph of nodes, then using `start()` and `stop()` functions to turn on and off the nodes at a given time, relative to the `AudioContext` clock. For ahead of time scheduling of events, this requires all future nodes to be realised. This is inefficient in terms of memory, but does give accurate timing. This appears to be a known issue that is being tracked by the development team.

So in this case, it is expected that users will try and implement their own event system. If this is the case, and nodes are used as-designed, it is possible to do this currently in Javascript via the `ScriptProcessorNode`. Scripts run inside these nodes *do* have access to the AudioContext, and so can create new nodes. However, timing is jitter-prone, as the ScriptProcessor is processed asynchronously from the audio thread. Also, the jitter is unbounded; the Javascript main thread can end up completely paused due to other processing or due to things like the page being backgrounded. Chris Rodgers has proposed a solution [Rodgers 2013], which is similar to the one proposed by Roger Dannenberg and Eli Brandt [Brandt and Dannenberg 1999]. However, this is not an accurate solution in that it does not guarantee reproducible results. It might be sufficient for many real-time scenarios, but not when processing may require sample-accurate timing. It is not appropriate for non-realtime scenarios.

As we have seen above, the new `AudioWorker` proposes Javascript-based processing code that is run synchronously in the audio thread. This would allow accurate event system to be written, but the problem is that in this case `AudioContext` is not available to the script run under this mechanism. That means even if you wrote a scheduler, you could not create nodes running in an AudioContext that is external to it. In this scenario, one is probably better off not using any of the nodes in WebAudio, and instead doing everything in Javascript. This abandons using any of the built-in nodes, but trades off for accuracy and reproducibility across browsers (which is not guaranteed with Web Audio code, see section 3). As noted above, there is an element of speculation in this discussion, however, as `AudioWorker` is only a specification at this moment. It is unknown whether the audio context will eventually be made available to AudioWorkers.

2.4. Offline rendering

As part of the current Editor's draft of the Web Audio specification, we see the presence of new audio context interface, represented by `OfflineAudioContext`. This is a welcome addition, which would allow non-realtime use cases to be addressed. It provides a means of running nodes asynchronously which are not dependent on the need of delivering samples in a given time period, so slow processes could be rendered through this method (and buffered for playback when needed). It writes the output of the process to memory (as an `AudioBuffer` object), and if the final destination of these is a file, then this has to be separately handled by Javascript and HTML5. It appears to provide much needed support for processing that is not designed for realtime audio. However at the time of writing, it is not possible to assess it in a more thorough way since it is still at a specification stage.

2.5. Extensibility

While support for Javascript-based extensions to the system exist, as discussed in section 2.1, there is no indication of plans or proposals for means of extending the system via natively-compiled nodes. Such components would be useful for two reasons: they would allow computationally-intensive processes to take advantage of implementation-language performance; and they would provide a simple means of porting existing code into web applications. Current estimates of difference between optimised Javascript code and native code plugins performing the same tasks indicate a slowdown by a factor of ten [Lazzarini et al. 2015], so the first point above is clearly justified. The second is similarly valid considering the wealth of open-source code for audio processing algorithms that exists in C or C++ forms.

It would be interesting, for instance, if the efforts that have been put in the Native Client (NaCl) [Yee et al. 2009] open-source project could somehow be incorporated into WebAudio via a well-defined interface maybe through a dedicated node. There has been some indication that this might work, as a user-level integration of the two via the script processor has been reported as functional, albeit with some significant issues, for instance in terms of added latencies in the audio path [Kleimola 2015]. The Portable Native Client (PNaCl) plugin system has been proved to be very useful for audio processing, for example, in one of the ports of the Csound system to the Web [Lazzarini et al. 2015].

One of the key aspects of the NaCl system is that it has been shown to be a secure way of extending Javascript applications [Sehr et al. 2010]. Given that many of the constraints to improving the support to lower-level programming in Web Audio appear to relate to security concerns, it appears that NaCl, in its PNaCl form, might provide a suitable environment for extensibility. The provision of an interface for NaCl could therefore provide a very powerful and secure plugin system for the API.

3. Implementation issues

The Web Audio API specification is implemented by browser vendors in different ways. Since the source code for the audio implementation does not stem from a unique upstream repository, such differences can be considerable. In order to explore this issue in a limited but detailed fashion, we have chosen to concentrate on a particular case study. We understand, from informal observations, that the differences discussed here may extend well

beyond this particular example. For instance, we have discovered that a certain browser (Safari) appears to apply a limit of -12dB for full scale audio, whereas other browsers, such as Chrome and Firefox, do not (allowing not only a 0dB full scale, but also not making any efforts to prevent off-scale amplitudes). However it is beyond the scope of this paper to provide a complete assessment of implementation issues. We have chosen two popular browser lines for this test, Google Chrome and Mozilla Firefox, which will provide a sample of the possible differences both in source code implementation and in sonic result.

3.1. Case study: the Oscillator node

In this case study, we have written a very simple Oscillator-based instrument consisting of an OscillatorNode connected to the output, in this case, producing a sawtooth wave:

```
var audioContext;
var freq = 344.53125,  end= 10, start = 1;
var oscNode = audioContext.createOscillator();
oscNode.type="sawtooh";
oscNode.frequency.value = freql
oscNode.connect(audioContext.destination);
oscNode.start(audioContext.currentTime + start);
oscNode.stop(audioContext.currentTime + start + end);
```

All signals had an $f_0 = 344.53125$, which at $f_s = 44100$ means 128 complete cycles in 16384 samples. This was used as the size of our DFT frame for analysis. The above program was run under the Chrome and Firefox browsers. We plotted the magnitude spectra for the sawtooth waves in figs 1 and 2 (Chrome and Firefox outputs, respectively), and their absolute difference in fig 3.

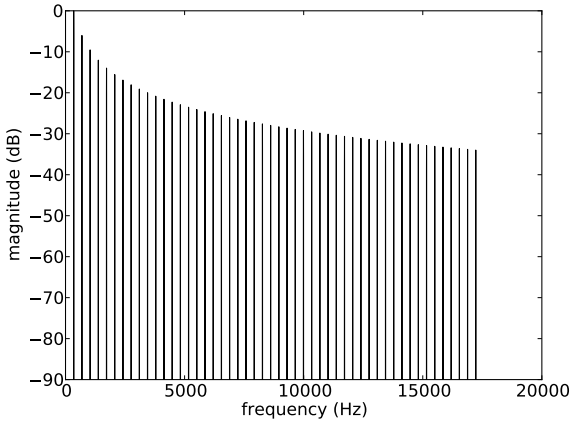

Figure 1. The magnitude spectrum of a sawtooth wave generated by Chrome

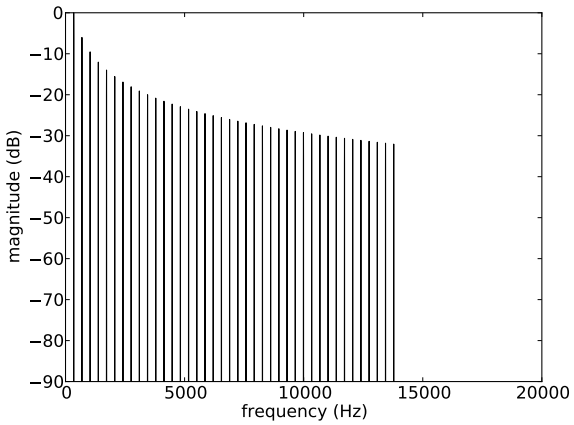

Figure 2. The magnitude spectrum of a sawtooth wave generated by Firefox

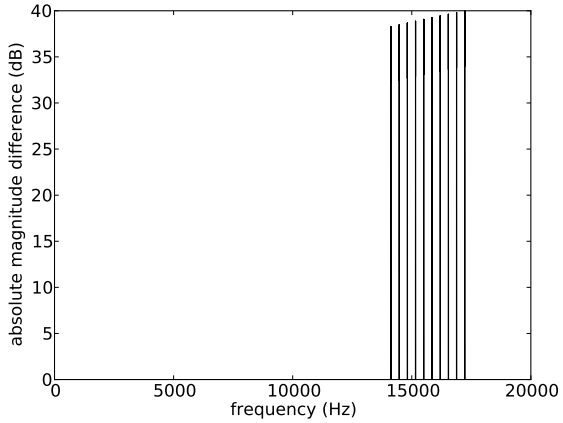

Figure 3. The absolute difference of the magnitude spectra of two sawtooth waves generated by Firefox and Chrome

In addition, we run the same program with `oscNode.type="square"` and plotted the results of the individual magnitude spectra in figs 4 and 5, as well as their absolute difference in fig 6.

From these plots, it is clear that at the high end of the spectrum, we have significantly different signals, as the Firefox output is quite drastically bandlimited, yield a difference of around 37-40dB between the two in the ten highest partials (sawtooth wave, five in the square wave case). Examining the source code for these two implementations of the Web Audio spec, we see that while the Chrome implementation uses a wavetable algorithm for implementing bandlimited versions of classic analogue waves, the bandlimited impulse train (BLIT) [Stilson and Smith 1996] method is used in Firefox. The Chrome implementation is much richer in harmonics, due to its use of three wavetables per octave over twelve octaves, which covers quite a lot of the spectrum up to the Nyquist frequency. In addition to the differences plotted here, we noticed the presence of a very low-frequency component (not visible in the figures above), which is present in the Firefox OscillatorNode signal as an artefact of the way BLIT is implemented.

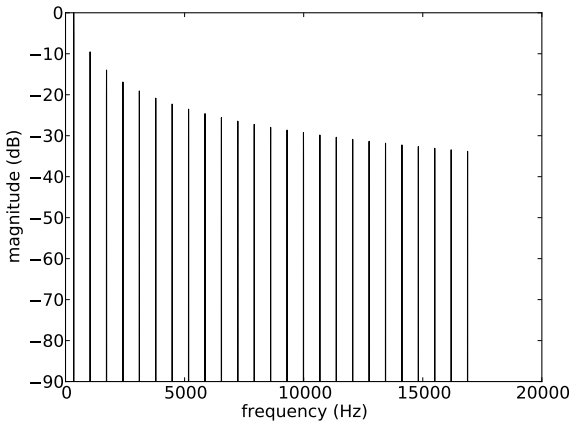

Figure 4. The magnitude spectrum of a square wave generated by Chrome

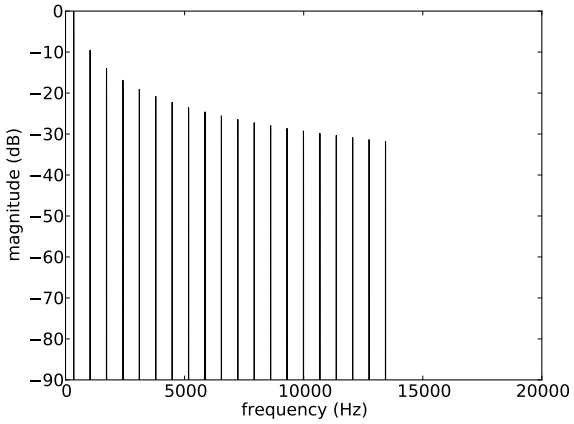

Figure 5. The magnitude spectrum of a square wave generated by Firefox

The differences discussed here stem from these implementations being, in sound and music computing terms, two clearly distinct unit generators. In a system such as Csound, with over 1,800 such components, they are assigned two different names (in this case, `vco` and `vco2`, also with slightly different parameters reflecting the particular methods used). The WebAudio specification is not definitive enough to prevent such deviations, and maybe not wide enough to accommodate them in a more suitable way. While we understand the desire to be succinct, we also note that the experience of the existing systems could have been used to inform the design of the API. Clearly, if we are to allow different implementations of bandlimited oscillators (and there are many of them), then we need to provide ways that users can distinguish between them. The development of Computer Music has been one in which precision and audio quality were always first-class citizens, and it is reasonable to expect these standards to be maintained in such an important software project.

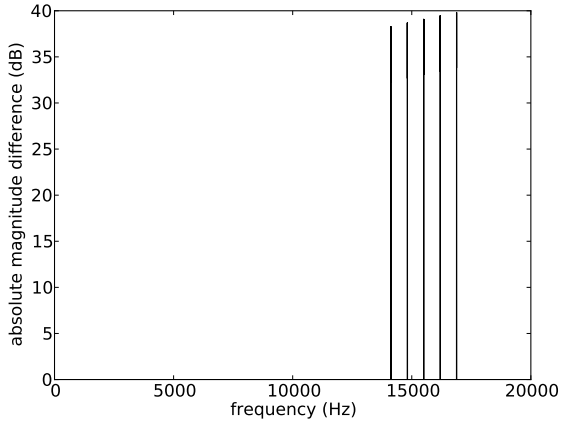

Figure 6. The absolute difference of the magnitude spectra of two square waves generated by Firefox and Chrome

As builtin nodes can differ, it is not possible to create consistent results across browsers. An alternative to this of course is to use Javascript-programmed audio code (either directly or via systems like Csound) to ensure the same results everywhere. It is also important to note that issues such as this are not confined to Web Audio, as differences in interpretations are not new to web applications. For instance, on the graphics side, browsers have long been known to render web pages differently (types, in particular, are an issue[Brown 2010]). However, this is widely acknowledged to be a less than desirable scenario.

4. Project directions

The Web Audio project is clearly a very significant project, which has been managed in an open way, through accessible code repositories, and a well-supported issue tracking system. Discussions on its directions have been carried out in open fora, and the main team members seem to take heed of user suggestions. On the other hand, the points made in this paper may indicate a certain lack of awareness of the fifty years of computer-based digital audio technologies. The history of computer music languages is rich in examples of interesting ideas and concepts [Lazzarini 2013], and these could be very useful to the design of WebAudio. Interestingly, developers seem to be well aware of commercially-available closed-source music software. Proprietary multitrack and MIDI programs Logic and GarageBand, for instance are name-checked in the Web Audio specification document[Adenot and Rodgers 2015], even though the functionality and use-cases of the API are closer to FLOSS music programming systems.

One way in which the project could take advantage of the wealth of ideas in FLOSS Computer Music systems is to develop a reference implementation for unit generators/nodes, based on source code that is openly available and well documented. This could be a way of addressing the issues raised in section 3, and a means of making good use of existing technology. Furthermore, a review of such systems could inform the decisions taken by the team in terms of steering the future directions of the API. Contributors to the discussion fora have already been bringing ideas that stem from academic research in the area, in an informal way. This could be enhanced by structured and systemic study that could be carried out as part of the development work.

5. Conclusions

The Web Audio framework is a very welcome development in audio programming, as it provides a number of potential applications that were previously less well supported. However, there are some key issues in its current implementation, and in its design, that need to be addressed, or at least, considered. On one hand, users of the framework should be made aware of these so that they can make informed decisions in the development process; on the other, developers might want to pay attention to the ones that can be addressed in some way. Our aim with this paper is to be able to contribute to the debate in the area of programming tools, so that support for a variety of approaches in music systems development is enhanced. From this perspective, we would like to offer the following summary of suggestions:

- The introduction of an *instrument* interface to enhance composability (section 2)
- Further flexibility for Audio Worker code (e.g. some form of access to the calling audio context) (2.1)
- New nodes, in particular one for handling control curve generation (2.2)
- More precise and flexible scheduling (2.3).
- Extensibility enhancements via native plugins (2.5).
- More precise definitions to minimise implementation differences (3).
- A reference implementation based on existing computer music systems (4).

References

Adenot, P. and Rodgers, C. (2015). Web Audio API, W3C Editor's Draft. `https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html`. Accessed: April 16, 2015.

Brandt, E. and Dannenberg, R. B. (1999). Time in distributed real-time systems. In *In Proc. Int. Computer Music Conference*, pages 523–526.

Brown, T. (2010). Type rendering: web browsers. Accessed: April 17, 2015.

Bélanger, O. (2015). PyO: dedicated Python module for digital signal processing. `http://ajaxsoundstudio.com/software/pyo/`. Accessed: April 17, 2015.

Ffitch, J., Lazzarini, V., Yi, S., Gogins, M., and Cabrera, A. (2015). Csound. `http://csound.github.io`. Accessed: April 16, 2015.

Hickson, I. (2014). Web Workers, Editor's Draft. `http://dev.w3.org/html5/workers/`. Accessed: April 18, 2015.

Ircam (2015). The 1^{st} Web Audio Conference . `http://wac.ircam.fr`. Accessed: April 16, 2015.

Kleimola, J. (2015). Daw plugins for web browsers. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Lazzarini, V. (2008). Interactive audio signal scripting. In *Proceedings of ICMC 2008*.

Lazzarini, V. (2013). The development of computer music programming systems. *Journal of New Music Research*, 42(1):97–110.

Lazzarini, V., Costello, E., Yi, S., and ffitch, J. (2014). Csound on the Web. In *Linux Audio Conference*, pages 77–84, Karlsruhe, Germany.

Lazzarini, V., Yi, S., Costello, E., and ffitch, J. (2015). Extending csound to the web. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Mann, Y. (2015). Interactive music with tone.js. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

McCartney, J. (2015). SuperCollider. `http://supercollider.github.io`. Accessed: April 16, 2015.

Monschke, J. (2015). Building a collaborative digital audio workstation based on the web audio api. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Puckette, M. (2015). Pure Data. `http://puredata.org`. Accessed: April 16, 2015.

Roberts, C., Wakefield, G., and Wright, M. (2013). The Web Browser As Synthesizer And Interface. *Proceedings of the International Conference on New Interfaces for Musical Expression*.

Rodgers, C. (2013). A Tale of Two Clocks. `http://www.html5rocks.com/en/tutorials/audio/scheduling/`. Accessed: April 17, 2015.

Sehr, D., Muth, R., Bifie, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B. (2010). Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*.

Stilson, T. and Smith, J. (1996). Alias-free digital synthesis of classic analog waveformss. In *In Proc. Int. Computer Music Conference*, page 332?335.

Wyse, L. (2015). Spatially distributed sound computing and rendering using the web audio platform. In *Proceedings of the Web Audio Conference 2015*, IRCAM, Paris, France.

Wyse, L. and Subramanian, S. (2013). The Viability of the Web Browser as a Computer Music Platform. *Computer Music Journal*, 37(4):10–23.

Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N. (2009). Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 IEEE Symposium on Security and Privacy*.