Lonce Wyse and Srikumar Subramanian
Communications and New Media Department
National University of Singapore
Blk AS6, #03-41
11 Computing Drive
Singapore 117416
lonce.wyse@nus.edu.sg
srikumarks@gmail.com

The Viability of the Web Browser as a Computer Music Platform

Abstract: The computer music community has historically pushed the boundaries of technologies for music-making, using and developing cutting-edge computing, communication, and interfaces in a wide variety of creative practices to meet exacting standards of quality. Several separate systems and protocols have been developed to serve this community, such as Max/MSP and Pd for synthesis and teaching, JackTrip for networked audio, MIDI/OSC for communication, as well as Max/MSP and TouchOSC for interface design, to name a few. With the still-nascent Web Audio API standard and related technologies, we are now, more than ever, seeing an increase in these capabilities and their integration in a single ubiquitous platform: the Web browser. In this article, we examine the suitability of the Web browser as a computer music platform in critical aspects of audio synthesis, timing, I/O, and communication. We focus on the new Web Audio API and situate it in the context of associated technologies to understand how well they together can be expected to meet the musical, computational, and development needs of the computer music community. We identify timing and extensibility as two key areas that still need work in order to meet those needs.

To date, despite the work of a few intrepid musical explorers, the Web browser platform has not been widely considered as a viable platform for the development of computer music. Professional-quality computer music platforms typically provide sample rates of 48 kHz and above, multiple channel configurations (e.g., 8, 7.1, 5.1), computational power to meet heavy signal processing demands, languages and compilers that can take advantage of the full computational power offered by a machine, the ability to handle standard communications protocols such as MIDI and Open Sound Control (OSC), reliable timing services capable of scheduling with sample accuracy, and the ability to deliver minimal throughput latency from gesture to sonic response. Access to local machine capabilities, such as file systems and media input and output, is essential. Real-time networked music performance requires reliable delivery of data between participants with minimal delay for both audio and control signals. Synthesis development systems must be extensible in addition to providing access to common units for constructing instruments and sound models. Special-purpose development environments are important for the creation of music and musically oriented applications.

Computer Music Journal, 37:4, pp. 10–23, Winter 2014
doi:10.1162/COMJ_a_00213

Why would musicians care about working in the browser, a platform not specifically designed for computer music? Max/MSP is an example of a platform specifically oriented towards musical design, composition, and performance that addresses stringent sound and musical quality demands, and that provides access to many key tools such as visual design and network communication "under one roof." Although the browser platform does not match the musically oriented features or native audio and interface performance of existing specialized tools, it does offer an attractive alternative for music creators in several respects. For example, the developer community is enormous and provides a wealth of libraries (many of them open-source) for everything from physics and graphics to user interface components, specialized mathematics, and many other areas relevant to computer music developers. Another advantage it offers the musical community is a highly developed infrastructure for supporting work requiring massive participation.

There is also a significant benefit the browser has to offer from the perspective of "users"—performers and actively participating audiences, as well as composers building on sounds or instrument designs contributed by others. Consider the following list of questions that arise in different scenarios for these users:

How do musicians access the sheet music they need for practice and performance?

What does an audience member need to do in a participatory piece of music in order to send data to performing musicians?
How does an ensemble of instrumentalists turn their mobile devices into the particular set of instruments necessary for a particular performance?
How would two remotely located musicians collaborate using a shared graphical score?
How would a composer find a patch he or she needs?
How would a Web designer access and use a synthesis component from an interactive sound developer?
What is necessary for two people to share a video connection and play instruments together at the same time?
What would 1,000 people need to do to "jam" together?
How do people use their phones to interact with a sound piece executing on their computers?

Currently, there are many answers to each of these questions, most of which involve some combination of exchanging physical media or downloading software (the right version for specific hardware), choosing directories, decompressing archived files, installing programs (using one of many different methods), setting up environment variables, installing the necessary related libraries, figuring out remote and local IP addresses, making sure a firewall is off, etc. If there were one simple answer to all these questions—"Point your browser to this URL"—this would have a significant impact on the usability and portability of computer music.

Even though the browser has historically been no match for the performance of thoroughbred computer music systems, musicians and developers did explore and push the boundaries of the creative possibilities the platform had to offer. William Duckworth's *Cathedral* (1997) is one of the earliest examples of interactive music and graphical art on the Web, and a multi-user version of Duckworth's *PitchWeb* debuted in 2001. JSyn provided a synthesis engine in the C language with a Java application programming interface (API) that could be used in browsers as a plug-in (Burk 1998). An early application of JSyn was a drum box allowing distributed users to jam together.

Recently, however, several new standards have been emerging from the World Wide Web Consortium (W3C)—in parallel with the specification of the hypertext markup language, revision 5 (HTML5)—that specifically address the media capabilities of the browser. The Web Audio API (Rogers 2012) is one of the new APIs that is of critical importance for computer music.

As of mid-2013, recent implementations of the major browsers are already capable of supporting a large variety of computer music practices, such as real-time audio synthesis applications, graphical interfaces, and simultaneous interaction between participants from around the world. Some examples include Gibber for "live coding" performance (Roberts and Kuchera-Morin 2012); WebPd, which supports graphical programming with a growing subset of Pure Data (Pd) functionality within the browser (McCormick and Piquemal 2013); and multi-player shared sequencers and instruments (e.g., Borgeat 2013). In this article, we evaluate the emerging browser platform for its potential to meet the whole range of diverse and more demanding needs of the computer music community. The Web Audio API provides facilities for low-level sound synthesis and processing, making it the most obvious enabling technology for computer music. A new synthesis library in itself, however, would be of minor importance if it were not part of the browser context. We will consider a variety of components of the new browser ecosystem that together are creating the disruptive potential for computer music making.

Overview

We begin our discussion with the development of Web technologies from a musical perspective. We then introduce a key new component of browser technology, the Web Audio API, and discuss its possibilities and limitations for synthesis applications with an eye toward compositionality and extensibility, followed by issues of latency, timing,

Table 1. Musically Related Web Technologies at a Glance

Web Audio API	Low latency, multi-channel, sample-accurate audio playback, and processing in the browser
Web MIDI API	Support for talking to connected MIDI devices and scheduled delivery of MIDI messages from within the browser
WebSocket protocol	Protocol for persistent two-way data exchange between browser and server; useful for workspaces shared over a Wide Area Network (WAN)
WebRTC	Peer-to-peer real-time communication between browsers for audio, video, and control data
Web Workers	Background processes for time-consuming computation
getUserMedia	Browser access to the user's microphone and webcam (part of WebRTC)
XMLHttpRequest	Fetch and post arbitrary data to a server without leaving or reloading a Web page; can be used to load audio samples for processing with the Web Audio API
Node.js	Server-side JavaScript engine with an asynchronous full network stack
WebGL	OpenGL-based stack for advanced two- and three-dimensional visualizations
Canvas 2D	JavaScript support for two-dimensional drawing and rendering in a browser
SVG	Scalable Vector Graphics document structure, renderable in browsers; useful for music notation
localStorage	Small-scale per-domain key/value storage for the browser, similar to cookies
FileSystem API	Per-domain persistent recursive file system for large data storage from within the browser

and synchronization. We address the communication needs of the computer music community with a look at the new WebRTC specification for peer-to-peer communication in browsers and how it compares with the current state-of-the-art system for audio communications, JackTrip (Cáceres and Chafe 2010). We conclude with some summary remarks about making and performing computer music within the browser.

Table 1 summarizes the browser-based technologies relevant to the computer music community, a subset of which we examine in detail.

Development of the Web from an Audio Perspective

Originally, the browser was deaf and mute, and it provided little in the way of interaction and dynamic behavior, beyond loading a new Web page from a remote server upon the specification of a web address or a mouse click on a hyperlink. In the late 1990s, most major browsers had added support for Java applets, which were built with all the capabilities of the general purpose Java language, though with "sandbox" security restrictions for protecting the local system. Java Sound, which was introduced as part of Java 1.3 in May 2000, supported real-time capture, processing, and generation of sampled sound in browser "applets" written purely in Java.

Several Java plug-ins were developed that were capable of high-quality interactive sound synthesis, such as JSyn (originally running on compiled native code and later on pure Java; cf. Burk 1998), and the pure-Java ASound (Wyse 2003). In the late 1990s, Flash, with its graphical and time-line oriented development environment, vector graphics, and object-oriented ActionScript language, became the standard plug-in platform for animation and interaction. It was not until 2008, however, that "dynamic sound generation" became available in Flash 10. Although usable for some animation and musical applications, latencies in the range of hundreds of milliseconds did not approach the much lower latencies provided by native applications such as Max/MSP.

Another important historical development came in 1995 with the introduction of the language JavaScript to the Netscape browser. JavaScript could be used by Web site developers to access components of Web pages and to program simple interactivity. Built-in support for Java on browsers has waned, and JavaScript is now the only general-purpose language built in to all browsers. Programs in other languages run in browsers on top of plug-ins that

support, for example, Flash or Java applets, but the plug-ins must be downloaded and installed by the user and run in environments that communicate with, but are separate from, the browser. The plug-in architecture also creates obstacles for the user and introduces security vulnerabilities. JavaScript, however, was originally designed neither for large-scale software projects nor for high-performance media applications.

JavaScript has evolved into a powerful general-purpose programming language with salient features that include its prototypal (rather than class) structure, its closure-based scoping rules, and the fact that its functions are first-class objects that can, for example, be created at run time—one of the language's more powerful and poetic features. JavaScript has some well-recognized design flaws (Crockford 2008), but the contribution of libraries from thousands of developers has significantly increased its usability. Several libraries have become de facto standards, such as jQuery for interacting with Web page components (Resig 2006), and RequireJS for supporting the modular development of large-scale software projects (Chung 2011).

JavaScript engines have been showing tremendous gains in performance since about 2008. The types of objects flowing through a JavaScript program cannot, in general, be predicted in advance for optimization purposes. Thus, the run times tend to be slower than native code. Modern browsers run on engines (Google Chrome on V8, Safari on Nitro, Firefox on IonMonkey) that use a wide variety of optimizations that compile code dynamically, or "just in time" (JIT). Current benchmarks comparing JavaScript with Java show JavaScript to run on average about three times as slowly as Java, while requiring on average about half the memory and half the code required for Java (Debian Project 2013). These "performance-enhancing" compilation strategies are significant for demanding interactive media applications, since JavaScript provides access to the all the musically related technologies listed in Table 1.

Further blurring the line between interpreted and compiled languages is asm.js, which is being developed at Mozilla (Herman, Wagner, and Zakai 2013). Asm.js is a subset of JavaScript that is amenable to ahead-of-time (AOT) compilation into efficient native code and that can be used as a target language for compilers of other languages such as C or C++.

The speedups due to JIT and AOT innovations in JavaScript engines apply to "pure" JavaScript code. When JavaScript APIs are defined for computational algorithms as part of the standards for Web browsers, however, those underlying algorithms are implemented in native code for each platform by the browser providers and do not need to be compiled at run time. This combination of efficient JavaScript execution and precompiled audio components defined by the new Web Audio API (discussed in detail subsequently) promises to have a significant impact on computer music possibilities within the browser.

Server-Side JavaScript

The Web operates largely on a client/server model, where clients run browsers that connect to servers serving data such as Web pages or real-time chat messages. Large, multi-threaded server applications, such as Apache, have historically been used for this purpose, and different server code components are often written in a variety of different languages (unlike typical client code).

Node.js (www.nodejs.org) is a software system for building lightweight servers using JavaScript and running on the V8 engine. Node.js has seen fast and accelerating popularity since its introduction in 2009. It uses an asynchronous method of calling functions, so that the calling code does not block while waiting for calls to complete. Possibly its most outstanding advantage over alternatives for music programmers is that code is written in JavaScript, the same language used by browser applications, permitting code reuse as well as reducing learning and development time.

Clients generally access servers over wide area networks (WANs). This raises the question of latency for musical applications. Latency greater than 10 msec between a gesture and a sounding result can disrupt a performer's practice, and latency greater than 25 msec between performers can disrupt

ensemble play (Cáceres and Chafe 2010). If a client and server are on opposite sides of the world, the speed of light limits the round trip from ever taking less than 130 msec.

Many musical applications can exist comfortably within wide area communication environments or hybrid environments where some communications are local (and thus fast) and others remote (and slower). Freeman (2010) explored live Web-based collaboration, and Canning (2012) used Node.js for a central server in a dynamic, shared musical-score environment over a WAN. For networked musical applications that have more demanding latency requirements, the client/server model can still work with the server running on a local area network (LAN), possibly even on the same machine as a client.

Local client/server architectures are a natural for many musical network applications such as those built on star-topology networks (Weinberg 2005) where client communications are coordinated through a hub. A LAN-based browser architecture could also be used, for example, to serve instrumental interfaces similar to TouchOSC (Fischer 2013) or Control (Roberts 2011) for real-time communication with synthesis code running on a server. If such systems were browser-based, clients could simply navigate to a Web page to access their interface (Roberts 2013; Wyse 2013 cf. "messageSurface"). This is both less complicated and less error-prone than manually downloading and installing applications. The browser/Node.js client/server structure is very flexible and can be run on one machine, a LAN, or a WAN, without changing a line of code.

Another significant aspect of networked musical applications is that client/server networking protocols have evolved considerably since the beginning of the millennium, when the primary browser communication model was based on a client "pulling" data from a server, using a hypertext transfer protocol (HTTP) request. This required a heavy overhead for establishing a connection for each request, only to have the connection immediately disappear after the request was fulfilled. Browsers used persistent HTTP connections behind the scenes to fetch multiple resources from a common source, but such a facility was not accessible to the client-side scripting layer. With the WebSocket API (Hickson 2011), it is now routine to establish persistent connections. These minimize the overhead for ongoing communication, such as subscription and push services, and allow the reception (and two-way exchange) of data not specifically requested. WebSockets are built on top of the transmission control protocol (TCP), which guarantees in-order and 100-percent packet delivery, though at the expense of higher latency. The Socket.IO library (Rauch 2013) makes it convenient to use WebSockets with similar code structure used on the server side in Node.js as well as in the client side.

The Web Audio API

The Web Audio API (Rogers 2012) is at the heart of the emerging technologies that support computer music in the browser. It is a JavaScript-API specification developed by the W3C and designed to support low-latency, sample-accurate playback and processing of audio in a browser. It is based on a "signal-flow graph" paradigm with nodes for synthesis and processing, and connections defining the flow of audio signals between nodes. This paradigm will be very familiar to users of the graphical programming languages Max/MSP and Pd. The components defined as part of the Web Audio API standard are implemented in optimized native code (typically C++) by browser platform providers. Thus, the components achieve very high-speed computation and low-latency throughput, as well as prolonged battery life in the case of mobile devices.

Our goal is neither to provide an introductory tutorial on programming with the Web Audio API (Smus 2011, 2013), nor to provide details on its internal workings. Instead, we provide a simple code listing to illustrate the basics of coding with the API (see Figure 1), and then dive into several specific issues of concern to prospective computer music community "power users" of the API, such as the ease-of-use, extensibility, and timing accuracy of the system.

Figure 1. Generating a decaying sine tone using the Web Audio API. This example illustrates the use of the AudioContext class, audio nodes, connections, and event scheduling.

```
var audioContext = new AudioContext();
var kFreq = 660, kDecayTime = 0.5, kStartTime = 1.5, kGain = 0.25;
var oscNode = audioContext.createOscillator();
oscNode.frequency.value = kFreq;
var gainNode = audioContext.createGain();
gainNode.gain.value = kGain;
gainNode.gain.setTargetAtTime(0.0, audioContext.currentTime, kDecayTime);
oscNode.connect(gainNode);
gainNode.connect(audioContext.destination);
// Start a little into the future.
oscNode.start(audioContext.currentTime + kStartTime);
// Stop when the sound decays by enough.
oscNode.stop(audioContext.currentTime + kStartTime + 12 * kDecayTime);
```

Synthesis

The specification of the Web Audio API provides for native components including oscillators, biquadratic filters, delay, dynamics compression, wave-shaping, one-dimensional convolution, an FFT-based spectrum-analyzer, audio-buffer sources, media-stream sources and sinks, and units for spatialization (including panning, Doppler shifting, sound cones, and head-related transfer functions). Some parameters can be used as sample-rate signals to control, for example, envelopes. The number of available units cannot compare to the 1,000 or so predefined in Max/MSP. Still, the increasing speed of JavaScript, the ability to script the signal-flow graph, and the availability of a ScriptProcessor node for expressing arbitrary audio-processing in JavaScript make it possible to create and encapsulate more complex structures into new JavaScript objects.

Compositionality (i.e., the ability to build larger building blocks using smaller ones) has played a central role in the design of many computer music systems, such as Max/MSP, SuperCollider (McCartney 2002), and ChucK (Wang and Cook 2003). In Max/MSP, for instance, composition is achieved by connecting objects using virtual wires and encapsulating a set of objects and their connections as a "patch," which can then on be treated as an object in its own right. In programming languages such as SuperCollider and ChucK, compositionality is achieved through functions. We consider structural compositionality in the next section; temporal compositionality is discussed later, as part of the section on "Scheduling, Synchronization, and Latency."

Structural Compositionality

The Web Audio API's architecture is a signal-flow graph that is, in the general case, compositional in nature. Topologically, subgraphs can be treated like nodes within a larger graph. Such composition requires stable nodes, however, and not all the native nodes provided by the API are stable. The Oscillator and AudioBufferSource are unstable source nodes since they are "single-use" only—that is, they can be started and stopped only once. After a stop() call has been issued, these nodes become useless. In fact, the specification recommends that implementations deallocate these nodes and the subgraphs that depend on them at appropriate times, preferably as soon as is possible. This behavior is referred to as *dynamic lifetime* in the specification (Rogers 2012). Although this feature may be technically efficacious and help with some fine-grained voice control, the disappearance of objects as a side effect of use will be unfamiliar to most programmers, as well as to the computer music community thinking in terms

of the flow-graph model common to Max/MSP and Pd. In addition, because source nodes are intended to be ephemeral, these nodes cannot serve at the boundaries of composition and must be encapsulated into stable nodes to restore compositionality to the overall system. We found such an encapsulation to be still possible to do by using JavaScript objects to mimic the connect/disconnect API of the native nodes (Subramanian 2012). With such encapsulation, it is possible to create JavaScript objects that model instruments having polyphonic voices, to which both global and per-voice effects can be applied.

ScriptProcessorNodes, or "script nodes," permit the execution of arbitrary JavaScript code to process or generate audio samples. These nodes can be made to satisfy the requirements for structural composition given some additional features implemented in pure JavaScript (Subramanian 2013b). These features, which include dynamic lifetime support and sample-accurate scheduling, enable script nodes to emulate the functionality provided by native nodes, albeit with lower performance and the introduction of additional delays.

Extensibility

We use the term *extensibility* to refer to the ability to create arbitrary new node types that, from an API user's perspective, behave in the same manner as native nodes (possibly with the cost of lower performance). Extensibility is important in computer music systems: First, to permit open-ended creative possibilities for synthesis, and second, to enable sharing of components within the community. Script nodes, which can execute arbitrary JavaScript code, satisfy the basic criterion for extensibility: Expressing arbitrary audio processing and generation in JavaScript.

The Web Audio API architecture gives the impression that script nodes and native modes are "equal citizens," but this is actually not the case, for a number of reasons. The most significant reason is that the native nodes are processed in a separate high-priority audio thread, whereas script nodes are processed in the Web page's main thread, along with other page-rendering and update events. Inter-process communication introduces latency and jitter. Furthermore, the native thread does not block to wait for script nodes to complete, which often means these script nodes must use larger buffer sizes than the native pipeline, to avoid audio glitches. The architecture is designed this way because arbitrary JavaScript cannot be permitted in the high-priority audio thread, for security reasons as well as to ensure low-latency, glitch-free audio when using the native nodes. Although alternatives, such as using "Web workers," have been proposed to process audio in JavaScript outside the main thread (Thereaux 2013), script nodes, for now, are likely to require instantiation with buffer sizes of 1,024 sample frames or more. The current implementations of the Web Audio API in WebKit browsers, such as Safari and Google Chrome, impose a two-buffer delay on script nodes—i.e., even a "pipe-through" script node would delay the audio by two buffer durations. The effect of this extra delay, and of the interruptions to audio due to page rendering and network events, is to prevent script nodes from serving as general-purpose components to extend the collection of native nodes with new node types, except in the cases where the delay is acceptable and where rich interactive graphics and low-latency networking are not required.

Wrapping the Web Audio API

There has been an explosion of commercial and developer-contributed libraries designed to hide underlying code complexity, to supply developers with convenient tools, and to provide a higher level API targeting the needs of a specific user group. One such library that wraps the Web Audio API to provide a sound designer interface is WAAX (Choi and Berger 2013). For example, it requires only a single line of code to create an AudioBufferSourceNode with a buffer filled from a file, rather than the some 16 lines it takes to achieve the same functionality using raw API calls. WAAX includes an integrated development environment, as well as straightforward ways of creating graphical user interface objects for controlling AudioParams.

The Web Audio API is a sound developer's interface rather than a general application developer's interface. Most application developers are not sound developers, but sound users, and they currently control sounds using a small handful of methods with names like "play", "stop", and, for three-dimensional audio, "setPosition" and "setListenerPosition". We have developed the jsaSound library (Wyse 2013 cf. "jsaSound") in recognition of the distinction between (1) interactive sound developers and (2) interactive sound users. It hides some of the complexity of using the Web Audio API (e.g., ephemeral source nodes), provides a set of tools for scheduling events (e.g., rhythmic patterns), and supports the creation of a consistent and simple API for all sound models. The API adds one key method to the familiar play() and stop() methods used for triggered sounds, and that is the setParameter() method that can be used with a parameter name or index as an argument, and that uses values in either parameter-specific ranges or the normalized range [0, 1].

Doing It All in JavaScript

Earlier, we mentioned the problems with combining script nodes with native nodes. These problems vanish, however, if all the audio work is done within a single script node, provided the main thread is mostly free for audio work. The disadvantage of this approach is that we get neither the functionality and speed advantages of the native nodes nor the latency benefits of running audio code in a high priority audio thread. We do, however, gain some interesting capabilities not possible with native nodes, such as single-sample-delay feedback systems and oversampling. Furthermore, script nodes may become more capable components in the future, using coding techniques that can be highly optimized, such as those defined in the working draft of asm.js (Herman, Wagner, and Zakai 2013). Given the current architecture, script nodes would still be subject to interruptions from other activity in the main thread, such as user interaction.

Modern JavaScript engines compile code to native code when the opportunity arises. Using eval(), JavaScript can serve as the target language for optimizing code generators that deal with higher levels of abstraction, such as audio-signal flow graphs. In this sense, JavaScript serves the same role that assembly languages do as the target for compilers of higher-level programming languages such as C, C++, or Objective-C. Roberts, Wakefield, and Wright (2013), for example, have developed a system called Gibberish, which exploits this technique.

We have now seen that, although it is possible to supplement the API in pure JavaScript to provide compositionality, the script node's limitations prevent the system from being extensible in ways computer musicians would demand, unless they are willing to forego native audio components. Next, we consider timing issues and synchronization with other computational components.

Scheduling, Synchronization, and Latency

The architecture implied by the specification of the Web Audio API raises some interesting problems for synchronization of other musical and visual activities with the audio processing. We begin our discussion with an examination of composing events "in time."

Temporal Compositionality

While structural compositionality is important for developing and building on sound synthesis techniques, temporal compositionality is central to the "music" in "computer music." Precise scheduling of events with reference to a common clock is necessary to ensure that temporal relationships between events desired by the composer or performer are not violated. Towards this end, the Web Audio API provides for sample-accurate scheduling of samplers, oscillators, and parameter curves.

Oscillator and AudioBufferSource nodes both have start() and stop() methods that take time (in seconds) as an argument and schedule their actions with sample-accuracy, provided the times are specified to be greater than audioContext.currentTime. AudioParams come in a-rate (audio rate) and k-rate

(control rate) varieties, providing values for each sample frame or at the start of each block of 128 sample frames, respectively. One powerful feature is that audio signals generated by nodes can also be used to control certain AudioParams, such as gain, frequency, and delay time, at the audio sample rate for native nodes.

Real-time interactive music systems require events to be scheduled as close as possible to "now." In general, the longer the delay, the worse the system's response will be. Programmers using the API for such applications will need to build their own schedulers (or use higher-level libraries that wrap the API) to queue up future events "just in time." For this to work well in the case of the Web Audio API, the weak scheduling facilities of JavaScript need to be used in conjunction with the sample-accurate scheduler provided by the Web Audio API, since the latter cannot be used to call back into JavaScript. Wilson (2013) discusses some of the real-time scheduling issues, and in particular, techniques for using timer callbacks to schedule events at short times into the future so that scheduling can remain sensitive to real-time input. The standard JavaScript timers—setInterval() and setTimeout()—are subject to large amounts of jitter, making them unsuitable for real-time interactivity. So the clock of choice for this purpose has become requestAnimationFrame() (Irish 2011), variants of which are available in most browser environments for callbacks at the time of each screen redraw. Another advantage of requestAnimationFrame() is that the callback is passed a timestamp corresponding to the time at which any visuals computed in the callback will be displayed. This is required for good audio-visual synchronization.

We have developed a Scheduler object for a library that uses the underlying sample-accurate scheduler along with requestAnimationFrame() to provide abstractions that separate the specification of temporal relationships between musical events from the performance of these specifications. This separation enables sample-accurate temporal composition. Our scheduler also supports tracks with their own tempo controls that can be scheduled to change over time by other tracks. Popular patterns such as temporal recursion can also be expressed with sample-accurate timing. The "Steller Explorer" features code illustrating such uses of this scheduler and is included in the "Steller" library (Subramanian 2013c).

Synchronization with Graphics

At the time of this writing, synchronization of audio with graphics can be achieved within a tolerance of one audio-buffer duration, which is the duration of each chunk of audio computed by an implementation of the Web Audio API. Ideally, this tolerance would be independent of the audio-buffer duration and be determined only by the time interval between displayed visual frames. Most desktop, iOS, and Android systems run their displays at 60 fps, which sets this tolerance at around 16 msec. Given a visual frame rate of 60 fps, good audio-visual synchronization requires that the audio time advances at least 120 times per second. This places an upper limit of 256 samples on the buffer size at a sampling rate of 44,100 Hz, assuming buffers need to be in sizes that are in powers of two. Because one of the goals of the Web Audio API is to bring low-latency audio capability to the web platform, implementations have thus far consistently chosen low buffer durations of 256 samples or less. The current draft of the standard, however, does not place upper limits on buffer durations in implementations.

For the synchronization tolerance to be independent of the audio-buffer duration, the ability to accurately map audio times to system times is required. Proposals have been made recently (Berkovitz 2013) for additional support to relate the audio time stamps based on "currentTime" to the high-resolution DOMHighResTimeStamp type (Mann 2012). This will also permit precise synchronization with MIDI systems, since the proposed Web MIDI API makes use of DOMHighResTimeStamps. Currently, precision of synchronization with MIDI is limited to the buffer duration of the implementation, which is not guaranteed to be identical across systems. This may lead to inconsistent experiences.

There is also a variable visual latency between issuing drawing commands and the scene appearing on the screen. This differs with the target technology used: Scalable Vector Graphics (SVG), Canvas 2D, or the Web Graphics Library (WebGL). Unfortunately,

we know of no way to find out about these delays using any of the APIs exposed via JavaScript. A possible workaround is to use device and configuration profiling in combination with a database of measured delays to achieve the necessary synchronization. The software application Tala Keeper (Subramanian 2013a), a visual metronome for the talas of South Indian classical music, provides an example of the scheduled synchronization possible with the current implementations of the Web Audio API.

Latency

Reliable, low-latency audio output is a key requirement for interactive music applications. As discussed by Kasten and Levien (2013) at the Google IO conference, language design, garbage collection, blocking and non-blocking code, threading, and shared memory access are all factors that influence the action-to-audio response time and require careful design of the audio subsystem. Here we present some indicative measurements of audio latency made on a current Web Audio API implementation and compare it with the state-of-the-art native performance running on the same hardware.

Comprehensive technical performance testing is beyond the scope of this article. To give some indication of the response-time capabilities of the browser platform, we tested the audio latency of the Google Chrome browser in response to (a) mouse clicks and (b) microphone input, and compared it with a native application (Max/MSP) running on the same hardware and operating systems. Three hardware platforms were used for testing: A MacBook Pro running Mac OS X, a MacBook Pro running Windows 8, and a Dell Laptop running Windows 7. (See the Appendix for details of the testing platforms.)

In each case, data were collected as two-channel audio, with one channel recording the input event (a mouse click or audio event) and the other channel recording the system output (either a synthetic noise burst in response to a mouse click, or a signal from the microphone passed through the system). Table 2 shows the results.

From Table 2 it can be seen that the native application latency is lower for both mouse and microphone input-to-sound latency (except when the default Microsoft OS drivers are used for the native application). The difference in throughput between the browser and native applications is vastly improved over historical plug-in architectures, however, and continues to improve.

Table 2. Latency for Native and Browser Applications

	Max/MSP	*Browser*
a. Mouse button click to sound out (msec)		
Mac OS X	29	36
PC Windows 7	71[1] / 49[2]	76
Mac Windows 8	99[1]	72
b. Microphone input to sound out (msec)		
Mac OS X	4	16
PC Windows 7	100[1] / 34[2]	55
Mac Windows 8	85[1]	54

Comparison of input-to-sound-output latency (in msec) for native (Max/MSP) and browser (Google Chrome) applications for mouse click and microphone input.
Notes: [1] *Using the default Microsoft OS audio driver.*
[2] *Using ASIO4ALL drivers.*

I/O and Communication Protocols

Computer music systems depend on fast and reliable input/output (I/O) facilities for parameter control, and collaborative musical performance requires networking facilities. The upcoming Web MIDI API provides access to controllers and synthesizers connected to the computer running the browser, and WebRTC provides for audio and video input and real-time peer-to-peer (P2P) communication capabilities. In this section, we discuss these technologies and compare the capabilities of WebRTC with JackTrip.

OSC (Wright 2005) has become a standard protocol for computer music communication, and it is supported by a large base of musical software. Although there are, unfortunately, no published plans for supporting OSC within the browser platform, there are open-source libraries for managing OSC that include creating the message format from other data structures, parsing incoming messages, and converting to and from user datagram protocol

(UDP) packets for network communication (Wyse 2013 cf. "json2osc").

MIDI capabilities are being developed in the form of the Web MIDI API (Kalliokoski and Wilson 2013), and are oriented toward providing access to connected MIDI devices for real-time input and output control (rather than toward rendering MIDI files). Though imminent, a full implementation of the standard has not yet been released at the time of this writing. Plug-ins already exist, however, that provide similar capabilities within the browser (e.g., www.jazz-soft.net).

WebRTC is another W3C standard in draft (Bergkvist et al. 2013) concerned with real-time communication (RTC). It is relevant to computer music for two different reasons. First, the API provides access to the video, microphone, and line inputs on the local machine. Access to local media streams has been restricted within browsers because of obvious security and privacy reasons. In the emerging standard, this is handled by requiring some form of user input to enable the access. The second way WebRTC is relevant for computer music is that it supports the P2P exchange of audio and video media over a local or wide-area network without the need to route streams through a central server (once the peer connection has been established). It thus addresses some of the same functionality currently provided to the computer music community by JackTrip (Cáceres and Chafe 2010), which was developed by the Soundwire group at Stanford University's Center for Computer Research in Music Acoustics.

The Web Audio API can work with the WebRTC method getUserMedia() to gain access to the media streams from the audio and video input (as well as from networked peers), and defines a MediaStreamAudioSourceNode type that wraps media streams. This node can be connected to other nodes in an audio graph just like any other AudioNode, allowing the signal to be further processed or analyzed.

Latency and reliability are critical issues for streaming audio in real-time performance environments. WebRTC was designed for P2P media communications for applications such as video chat. For network communication, WebRTC defaults to using UDP (which has faster but less reliable packet delivery than does TCP), but it falls back to other protocols if UDP fails for any reason. (There is also a separate channel for "signaling" meta-information about the communication.) The current open-source implementation of WebRTC (www.webrtc.org) supports only one-to-one communication, but libraries already exist for richer networks. WebRTC also has a built-in packet loss scheme and manages network address translation.

WebRTC uses the Opus audio codec by default. Opus uses a lossy compression scheme and supports multiple channels, variable and constant bit rates up to 510 kbits/sec, sample rates up to 48 kHz, and frame sizes down to 2.5 msec. These specifications meet the needs of many music applications, but do not reach JackTrip's quality. JackTrip supports uncompressed transmission (which is thus lossless and has no delay due to a compression algorithm), has sub-millisecond frame sizes (and concomitantly small packet sizes), and uses redundant streams to minimize the packet loss characteristic of UDP.

WebRTC includes an RTCDataChannel API. This supports the P2P exchange of arbitrary data that can be used in conjunction with audio and video. The data channel could be used to send OSC data, for example, which could control sound synthesis in synchronization with the audio and/or video streams being received. This assumes access to the necessary time-stamp data will be available in future versions of this API.

Now that we have examined most of the major individual pieces of the browser technology ecosystem that are relevant to computer music, we will take a step back to offer some concluding remarks on the platform as a whole.

Summary and Discussion

Artists using the computer to make sound and music draw on a wide variety of tools to address their highly demanding needs. These diverse needs include high-quality sound synthesis, instrument and interface design tools, robust low-latency and low-jitter communications, musically oriented development environments, interaction between devices over local and wide area networks, and

high-quality multi-channel synthesis. Different creative goals make different tools appropriate in different contexts.

The browser offers some unique capabilities for musicians due to its natural connectedness, its huge developer community, and the ease of access and portability of work it provides. It is true that important forays into the use of the browser for music have been made for over 15 years. For the platform to become central to the computer music community, however, not only will it have to be useful for some composers exploring certain kinds of music, but it will have to approach the performance of many different special-purpose tools that musicians use today on each of many different fronts. That is what we mean by "viable."

Emerging browser standards from WC3, such as the Web Audio API and WebRTC, are key enablers because they are bringing native computational speed and access to media resources to the platform capabilities. JavaScript, the language of browsers, has become much faster, has been extended with libraries that vastly increase its usability, and can now be used for coding both server- and client-side applications.

On some fronts the gap between the browser and native platforms has closed: The browser can support music with multiple channels and reasonably high sample rates, it uses a language that can run demanding audio code, and it provides (or will soon provide) access to local I/O channels for MIDI, video, and audio. On others fronts—such as OSC support, graphical programming languages, and input-to-output latency—the gap is still open, but appears to be closing. Finally, there are some "pain points" that remain obstructions to widespread adoption of the platform for the computer music community, and the path to a solution is not at all clear. First, as long as audio is susceptible to "glitching" because of a user-interface or garbage-collection event, concert-quality computer music performance will be impossible on the platform. Second, significant inherent limits on the extensibility of the synthesis engine will also prevent its widespread use among sound and instrument designers, who have excellent alternative platforms for their creativity. No alternatives to the beleaguered, general-purpose extension mechanism currently available—ScriptProcessorNodes—are visible on the horizon.

There are strong incentives for developing music systems in the browser environment, and the community of Web developers is huge. For that reason, we expect the growth in capabilities to continue apace, for example, through developer-contributed libraries. Two critical outstanding issues we have identified—timing and extensibility—need to be addressed by the developers of core standards and systems, in conjunction with the providers of operating systems and browsers, before the platform will be viable for the most demanding computer music applications. Nonetheless, the emerging capabilities of the browser already offer many new creative possibilities for musicians to explore.

Acknowledgments

This work was supported by a Singapore MOE grant FY2011-FRC3-003, "Folk Media: Interactive Sonic Rigs for Traditional Storytelling." Thanks to Gerry Beauregard for his insights and assistance in testing. Deep gratitude goes to Chris Rogers, previous editor of the W3C Web Audio API, for valuable and detailed feedback on the draft of this article (as well as the tireless effort he and the other W3C Audio Working Group members have put in to developing the new specification and implementations). We would also like to acknowledge the important contribution made by three peer reviewers who were clearly drawn from the pool of researchers and musicians we most admire.

References

Bergkvist, A., et al. 2013. "WebRTC 1.0: Real-Time Communication between Browsers." Available online at www.w3.org/TR/webrtc. Accessed August 2013.

Berkovitz, J. 2013. "W3C bug report #20698, comment 23." Available online at www.w3.org/Bugs/Public/show_bug.cgi?id=20698#c23. Accessed June, 2013.

Borgeat, P. 2013. "Interactive Networked Web Audio Experiences." *Network Music Festival*. Available online at networkmusicfestival.org/2013/interactive/. Accessed July 2013.

Burk, P. 1998. "JSyn: A Real-Time Synthesis API for Java." In *Proceedings of the International Computer Music Conference*, pp. 252–255.
Cáceres, J., and C. Chafe. 2010. "JackTrip: Under the Hood of an Engine for Network Audio." *Journal of New Music Research* 39(3)183–187.
Canning, R. 2012. "Real-Time Web Technologies in the Networked Performance Environment." In *Proceedings of the International Computer Music Conference*, pp. 315–319.
Choi, Hongchan, and J. Berger. 2013. "WAAX: Web Audio API eXtension." In *Proceedings of New Interfaces for Musical Expression*, pp. 499–502.
Chung, A. 2011. "RequireJS: A JavaScript Module Loader." Available online at requirejs.org. Accessed June 2013.
Crockford, D. 2008. *JavaScript: The Good Parts*. Sebastopol, California: O'Reilly.
Debian Project. 2013. "JavaScript Benchmarks." Available online at benchmarksgame.alioth.debian.org/u32/benchmark.php?test=all&lang=v8&lang2=Java&data=u32. Accessed June 2013.
Duckworth, W. 1997. "Cathedral." Available online at www.billduckworth.com/cathedralproject. Accessed August 2013.
Fischer, R. 2013. "TouchOSC." Available online at hexler.net/docs/touchosc. Accessed June 2013.
Freeman, J. 2010. "Web-Based Collaboration, Live Musical Performance, and Open-Form Scores." *International Journal of Performance Arts and Digital Media* 6:(2)149–170.
Herman, D., L. Wagner, and A. Zakai. 2013. "asm.js: Working Draft 17 March 2013." Available online at asmjs.org/spec/latest. Accessed June 2013.
Hickson, I. 2011. "The WebSocket API: W3C Working Draft." www.w3.org/TR/2011/WD-websockets-20110419. Accessed June 2013.
Irish, P. 2011. "requestAnimationFrame for Smart Animating." www.paulirish.com/2011/requestanimationframe-for-smart-animating/. Accessed June, 2013.
Kalliokoski, J., and C. Wilson. 2013. "Web MIDI API." Available online at webaudio.github.io/web-midi-api. Accessed August 2013.
Kasten, G., and R. Levien. 2013. "Google IO Presentation." Available online at developers.google.com/events/io/sessions/325993827. Accessed June 2013.
Mann, J. 2012. "High Resolution Time: W3C Recommendation." Available online at www.w3.org/TR/hr-time/. Accessed June 2013.
McCartney, J. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal* 26(4):61–68.
McCormick, C., and S. Piquemal. 2013. "WebPd" Available online at github.com/sebpiq/WebPd. Accessed July 2013.
Rauch, G. 2013. "Socket.IO." Available online at socket.io. Accessed June 2013.
Resig, J. 2006. "jQuery: Write Less, Do More." Available online at jQuery.com. Accessed June 2013.
Roberts, C. 2011. "Control: Software for End-User Interface Programming and Interactive Performance." In *Proceedings of the International Computer Music Conference*, pp. 425–428.
Roberts, C. 2013. "interface.editor." Available online at github.com/charlieroberts/interface.editor. Accessed June 2013.
Roberts, C., and J. Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference*, pp. 64–69.
Roberts, C., G. Wakefield, and M. Wright. 2013. "The Web Browser as Synthesizer and Interface." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 313–318.
Rogers, C. 2012. "Web Audio API: WC3 Editor's Draft." Available online at dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html. Accessed June 2013.
Smus, B. 2011. "Getting Started with Web Audio API." www.html5rocks.com/en/tutorials/webaudio/intro. Accessed June 2013.
Smus, B. 2013. *Web Audio API*. Sebastopol, California: O'Reilly.
Subramanian, S. K. 2012. "GraphNode." Available online at github.com/srikumarks/steller/blob/master/src/graphnode.js. Accessed June 2013.
Subramanian, S. K. 2013a. "Tala Keeper." Available online at talakeeper.org/talas.html. Accessed June 2013.
Subramanian, S. K. 2013b. "Taming the ScriptProcessorNode." Available online at sriku.org/blog/2013/01/30/taming-the-scriptprocessornode. Accessed July 2013.
Subramanian, S. K. 2013c. "Steller Explorer" Available online at sriku.org/demos/steller_explorer. Accessed July 2013.
Thereaux, O. 2013. "Resolution at the 2013-03-27 f2f meeting." Available online at www.w3.org/Bugs/Public/show_bug.cgi?id=17415#c94. Accessed June 2013.
Wang, G., and P. R. Cook. 2003. "ChucK: A Concurrent, On-the-Fly Audio Programming Language." In *Proceedings of the International Computer Music Conference*, pp. 219–226.
Weinberg, G. 2005. "Interconnected Musical Networks: Toward a Theoretical Framework." *Computer Music Journal* 29(2):23–39.

Wilson, C. 2013. "A Tale of Two Clocks—Scheduling Web Audio with Precision." Available online at www.html5rocks.com/en/tutorials/audio/scheduling. Accessed June 2013.
Wright, M. 2005. "Open Sound Control: An Enabling Technology for Musical Networking." *Organised Sound* 10(3):193–200.
Wyse, L. 2003. "A Sound Modeling and Synthesis System Designed for Maximum Usability." In *Proceedings of the International Computer Music Conference*, pp. 447–451.
Wyse, L. 2013. "Software Projects." Available online at anclab.org/software/software.html. Accessed June 2013.

Appendix

Latency testing was performed using a 44.1kHz sampling rate on: (1) a Dell XPS laptop, Windows 7, Intel Core 2 Duo at 2.4 GHz, and (2) a MacBook Pro, 15-in., mid 2010, Mac OS X 10.8.3, Intel Core i7 at 2.66 GHz.

Both hardware platforms were running Max 6.0 with the following audio settings:

Driver: Ad_directsound (default operating system) driver
Thread priority: Highest
Latency: 50 msec
IO vector size: 32
Signal vector size: 32
CNMAT units for OSC parsing

For the ASIO driver configuration:

Driver: ASIO4ALL
I/0 vector size: 512
Signal vector size: 2

For the browser tests: Google Chrome version 27.0.1453.93

All measurements are based on five trials. Variance was negligible and is thus not reported here.