Synchronisation for Distributed Audio Rendering over Heterogeneous Devices, in HTML5

Jean-Philippe Lambert
UMR STMS
IRCAM-CNRS-UPMC
Sorbonne Universités
UPMC Univ Paris 06
lambert@ircam.fr

Sébastien Robaszkiewicz
UMR STMS
IRCAM-CNRS-UPMC
Sorbonne Universités
UPMC Univ Paris 06
robaszkiewicz@ircam.fr

Norbert Schnell
UMR STMS
IRCAM-CNRS-UPMC
Sorbonne Universités
UPMC Univ Paris 06
schnell@ircam.fr

ABSTRACT

The HTML5 standard is wide-spread on mobile devices. In combination with the Web Audio API, it allows for massively distributed real-time audio rendering. But timing issues exist, mainly because of the lack of standard inter-device synchronisation. This paper proposes a synchronisation solution based on HTML5. Using a shared reference time, we achieved the distributed rendering of audio events with an individual accuracy of 1 to 10 ms, 5 ms in standard deviation, which is more accurate than the audio block duration, for any device that we measured.

CCS Concepts

•**Applied computing** → **Sound and music computing;** •**Networks** → *Time synchronization protocols;* •**Computer systems organization** → Distributed architectures; Real-time systems;

Keywords

Synchronisation, HTML5, Web Audio API, Audio, Distributed

1. INTRODUCTION

The generalisation of mobile devices that embed facilities for real-time audio, processing, motion sensors, and networking, has made it easy to use them for a wide range of applications. The fifth major revision of the Hypertext Markup Language (HTML5) [18] grants access to these features in a Web page from a standard Web browser, instead of developing native applications. Massively distributed applications are now within the reach of anyone, and this is of particular interest for audio applications, as one can distribute a complete audio process to a mobile device, up to an integrated speaker.

However, various platforms exhibit various behaviours. First, the devices are very heterogeneous in their hardware and software capabilities. They are also subject to complex and live interactions of the system and the user, such as software updates, messaging, and energy-saving. Finally, the different browsers provide different support of the standard [2, 19], that is wide-spread but still evolving.

In a situation that brings a lot of participants to the same physical space, like in a concert, the organisation of sounds in time and space is important. In this paper, among the various sound qualities, we focus on the timing in such a situation. There are different orders of magnitude for accuracy, depending on the usage. For multi-channel rendering, we need to be sample-accurate, which translates to around 20 µs for a common sample-rate. The human threshold of sensitivity is 1 ms for clicks and trained listeners [5, 6]. Two coherent speech sounds of the same level appear as echoes when their difference of time is more than 30 ms [1, p. 224], and the accuracy of the violin section of an orchestra is around 40 ms [15, p. 86]. While the sample-accuracy is beyond our reach for now, we aim at the perceptive coherency of distributed sound events. To achieve that, we need synchronisation.

In an orchestra, the conductor provides the synchronisation, while for most specialised audio devices, a cable transmits the word clock. However, mobile browsers can not use a cable, nor a standard facility for a reference time or synchronisation, yet [11]. The synchronisation of clocks and devices is not a new problem, and the Network Time Protocol (NTP) [7] is a well-established solution used by millions of servers [8]. While it requires a specific server, it has proved to be robust over the years.

Our contribution is based on the HTML5 standard, with a focus on real-time audio rendering, with the Web Audio application programming interface (API). We target any user-agent that supports the standard. Within this context, we propose a pure JavaScript (JS) solution, that runs in any standard browser, or server. We track the timing issues in Section 2, then we propose a synchronisation solution in Section 3, and we evaluate the resulting implementation in Section 4. The relevant source code is libre.[1]

2. LATENCY

We conducted a simple experiment at the Web Audio Conference (WAC) 2015. On a laptop, we ran a Node.js server, and we set-up a local WiFi network with an external router. Every second, the server sent a message via a WebSocket on two connected devices: a reference device and a test device.

Web Audio Conference WAC-2016, April 4–6, 2016, Atlanta, USA.

[1]The source code used for clock synchronisation is published at `https://github.com/collective-soundworks/sync/`, and the source code used for device calibration and measurements is published at `https://github.com/collective-soundworks/soundworks-calibration/`.

The devices played the sound with the Web Audio API as soon as possible upon receipt of this message. With both devices at the same distance of the ears (or of the microphones), one could manually adjust the delay of the test device, to try to play simultaneously with the reference device. We experienced a lot of jitter for some devices, especially noticeable for those with a delay over 200 ms: in this case, we selected an approximate mean delay value over time.

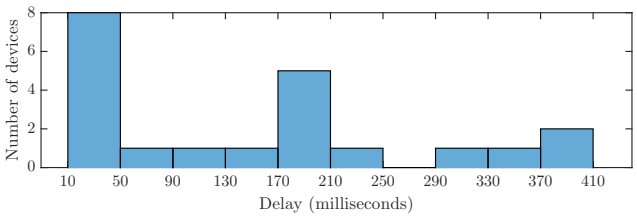

Figure 1: Histogram of relative overall delays (including network and audio) of various devices. Android 4.1, 4.2, 4.3, 4.4, 5.0; FirefoxOS 2.1; iOS 7, 8.

Figure 1 shows the histogram of these relative delays. While the values are specific to this experiment, they give an order of the magnitude of the problem: around 400 ms of relative time difference across the extrema.[2] And, for precise timing, we also need to address the jitter.

From our previous example, we track the delays, and note them with arrows on Figure 2. We call *host* the software that is either a browser or a server, that runs in an operating system (OS), and that allows to run our *application* code.

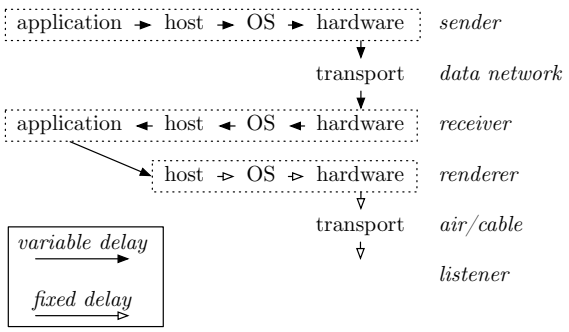

Figure 2: Accumulation of multiple delays from a remote *sender* to a *listener*.

On the server, we wait until the host schedules our *sender*'s code for execution. Then, we send a message via the WebSocket API, that goes through the host, the OS, and the hardware, before leaving the server. After the transport, it arrives to the *receiver*'s hardware, OS, and host, before our code is called. Then, the *renderer*'s code schedules a sound via the Web Audio API, to the next audio block. Then the host outputs the sound through the OS, and the hardware. In the air, the transmission delay depends on the distance of the listener.

We split the problem in two parts. From the *sender* to the *receiver*, we use a shared synchronised time, in order to compensate the total variable delay (see Section 3); in the *renderer*, we calibrate and compensate the fixed audio latency (see Section 3.2); and we keep a constant distance from the audio output, or we use a cable, to ignore the position of the *listener*.

[2]It is a lot: in the air, sound travels 130 m during this time.

3. SYNCHRONISATION

With a shared reference time, we can compensate any delay from a *sender* to a *receiver*, providing that the *sender* anticipates the total accumulated delay.[3] For example, one can use a look-ahead scheduler [22] based on a reference time. It is also possible to use multiple schedulers, all based on the same reference time, even on distributed devices. The question is how to get a shared reference time.

While we follow the same principles as NTP, we do not use it directly for several reasons. First we need a solution that works without a specific server, as our reference possibly comes from a browser. And we adapt the solution to our particular context, Web pages, that need a quick start-up and stabilisation (in seconds, or minutes, instead of hours for NTP). Finally, we integrate all the constraints inherent to the HTML5 environment.

3.1 Variable Latency

We follow the principles of NTP [14]: in a continuous process, we exchange the times between a client and the reference, we filter the transfers to keep only the quickest ones [10] and we estimate the reference clock, according to a local clock. It is a continuous process, as every clock varies over time.

3.1.1 Clocks

We need a precise reference clock, to get a precise reference time for scheduling audio events. In an HTML5 environment, `performance.now` is adapted,[4,5] while `Date.now` should only be used as a last resort.[6]

Within an `AudioContext`, it is possible to schedule events according to an absolute `currentTime` [16] with a resolution of 1 sample. However, as the standard does not requires it to be *strictly* monotonic, `currentTime` advances by steps that correspond to the audio block duration. Then, to measure or schedule any event relative to an other time reference like `performance.now`, or to an external event, its precision is as low as the audio block duration.[7] The mapping of `currentTime` to `performance.now` in the standard will hopefully solve this issue for any compliant browser [21].

While `currentTime` is a bad choice as a reference clock, it is a good candidate for a local clock, as we ultimately use it in order to render audio with the Web Audio API. The synchronisation process that we propose does not imply the

[3]as there is no way to reduce the intrinsic latency

[4]The resolution of `performance.now` is 1 µs, as it returns a `DOMHighResTimeStamp`, and it is monotonic. However, *"If the User Agent is unable to provide a time value accurate to a thousandth of a millisecond due to hardware or software constraints, the User Agent can represent a DOMHighResTimeStamp as a number of milliseconds accurate to a millisecond."* [17].

[5]`performance.now` does not exist for our server that runs Node.js, but we then use `process.hrtime`, as it is monotonic with a resolution of 1 ns [12].

[6]`Date.now` resolution is 1 ms [3], and it is not monotonic, as it follows the OS time, with its adjustments.

[7]While the audio block duration is not directly available in HTML5, we measured it by requesting a series of time and by looking at the advances. We measured values from 2.9 to 93 ms.

use of the audio clock,[8] but is tolerant to it. Moreover, by directly estimating the reference time from the `currentTime`, we maximise the resulting audio scheduling accuracy. From now, we focus on the audio clock as the local clock and on `performance.now` as the reference clock.

3.1.2 Ping-Pong Scheme

The algorithm is developed around a ping-pong scheme, where a client sends a time-stamp t_{ping} to a reference, that applies its own time-stamps: T_{ping} on the reception, and T_{pong} on the emission of its reply. The client then applies a final time-stamp t_{pong}.[9]

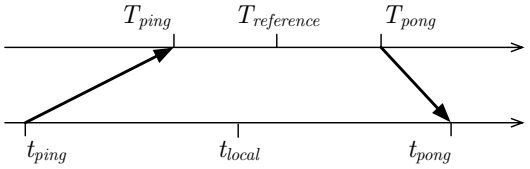

Figure 3: Exchange of time-stamps during a ping-pong round-trip.

T is used for reference time, while t is used for local time. Given that both times are expressed with the same unit, this allows the following estimations:

$$T = T_{reference} = (T_{pong} + T_{ping})/2 \quad (1)$$

$$t = t_{local} = (t_{pong} + t_{ping})/2 \quad (2)$$

$$\mathit{offset} = t_{local} - T_{reference} \quad (3)$$

$$\mathit{travel}_{duration} = t_{pong} - t_{ping} - (T_{pong} - T_{ping}) \quad (4)$$

This relies on the hypothesis that the network delay is symmetrical, which is never exact, but acceptable when the transmission delay is short, because it bounds the introduced bias. The design of the algorithm favours that.

First, instead of a single ping-pong probe, we send a series of them and we keep only the ones with the shortest $\mathit{travel}_{duration}$.

To minimise the network congestion, we only send a ping after a pong arrived, or after a time-out. Moreover, the time-out duration increases on any delayed pong (and decreases on arrival, for responsiveness). Finally, there is a random delay in-between the ping-pong series, to spread over time the network usage of the multiple clients.

There is also an implication of our use-case, that brings several mobile devices in the same physical space: we use a wireless *local* area network (WLAN), for direct connection between devices.

3.1.3 Estimation of the Reference Time

We assume that the reference clock T can be expressed linearly from the local clock t with an offset T_0 and a frequency ratio R, like NTP.

$$T(t) = T_0 + R(t - t_0) \quad (5)$$

To directly use the audio time, as the local time to synchronise with the reference time, we need to take care about its low precision. To compensate that, the time-span of the observations must we long enough for an accurate estimation of R. On the other hand, it should be short enough to quickly adapt to varying clocks, and to start. One last consideration is that we need enough samples to be able to accurately estimate t_0, in order to get a more precise reference than the audio block duration. To mitigate these constraints, we split our synchronisation process in two stages, so as it starts quickly, and it then uses long-term data for better accuracy.

During the *training stage*, we only adjust the *offset* of the local clock, after each ping-pong series. We take the mean *offset* over the 3 quickest $\mathit{travel}_{duration}$ of the last series, to compensate for jitter.

$$T(t) = t - \mathit{offset} \quad (6)$$

The heuristic driving the transition to the *synchronised stage* is that the estimation of the reference clock, that uses R, should be at least as accurate as in the training, despite the errors on R and t_0.[10] Then, given a set of measurements (t_i, T_i), obtained after the ping-pong series and the filtering, we estimate R with a simple linear regression, which is adapted for a real-time context, as it requires little processing [4].

$$R = \frac{\mathrm{Cov}[t,T]}{\mathrm{Var}[t]} = \frac{\overline{t.T} - \bar{t}.\overline{T}}{\overline{t^2} - \bar{t}^2} \quad (7)$$

While $T_0 = \overline{T}$ and $t_0 = \bar{t}$ are computed at the same time.

As the process is continuous, it is important to monitor it, for reporting purposes, but also to react on unexpected changes. We go back to the training stage any time the estimation of R deviates too much.[11]

3.2 Fixed Audio Latency

There is no standard way to query the additional latency added after our *renderer* code (see Figure 2) schedules audio via the Web Audio API, yet [20]. We also measured that guessing the audio block duration is not providing enough information to determine the total added latency. So the only choice for now is to measure it, for any type of device (software and hardware), in order to compensate it later.

We run two audio processes that use a shared synchronised clock. One, that is always the same and that we chose because it is the most stable, serves as a reference. In rendered audio, we measure the difference in time of the device to calibrate from the reference. (See Section 4 for the measurements.)

To get per-device calibration, we store it on the device with `localStorage`. We also store it on a server, to be able to reproduce a calibration on devices of the same type, with the user-agent[12] as the fuzzy key[13] of the data-base. Of course, this introduces inaccuracy.

[8] `performance.now` is a perfectly adapted local clock, too.

[9] It is not possible to issue an accurate time-stamp within the browser environment, because it is set in the JS event loop and not by the hardware when it is actually sent or received [9]. Then, the transport duration *de facto* includes the variable duration between our JS code and the hardware (see Figure 2). Moreover, if we use the low precision audio clock to issue a time-stamp, its precision is also low.

[10] After measuring devices with a block duration of 5.8 to 85 ms, we chose the following values as a good trade-off between speed, memory, computation, and the necessary data for an accurate estimation: series of 10 ping-pong probes occur every 10 to 15 s; the training lasts for 2 min; the linear regression is computed over a time-span of 15 min, on the quickest $\mathit{travel}_{duration}$ of each series.

[11] We use 500 PPM, which is 0.05 %, or the precision of an old mechanical clock [13].

[12] While it is fine for most phones and tablets, it provides no hardware information for the desktop or laptop devices.

[13] In order to accommodate to the variations and to still be able to apply the calibration to most devices, we use a

Although this works in practice, it is long, error-prone, and an endless effort to integrate new devices and updates, so the integration of such a feature to the standard is eagerly waited.

4. MEASUREMENTS

On a desktop computer, a Node.js server is running to provide a reference time. It also emits a request each second, for any client to output a click scheduled one second in the future, according to the current reference time. On the same computer, a multi-track recorder records the sound outputs of all the browsers, local or connected via the WLAN.

The time of each measurement starts with the synchronisation processes.

Device	OS	Browser
Galaxy A3 (mobile)	Android 5.0.2	Chrome 45
Galaxy S3 Mini (mobile)	Android 4.2.2	Chrome 44
iMac (desktop)	OS X 10.9.5	Chrome 45
iPad Mini 3 (mobile)	iOS 9.0.2	Safari 9
iPhone 4 (mobile)	iOS 7.1.2	Safari 7
iPod touch 5 (mobile)	iOS 8.4.1	Safari 8
MacBook Pro (laptop)	OS X 10.9.5	Firefox 41
Nexus 4 (mobile)	Android 5.1.1	Chrome 45
StarAddict III (mobile)	Android 4.2.2	Chrome 39
XPS 12 (laptop)	Windows 10.0	Edge 12

Table 1: Devices used for measurements.

In Table 1 we list the devices that we used for this set of measurements. From now on, to refer a particular device type, we only use a distinct letter arbitrarily chosen.

4.1 Homogeneous Devices

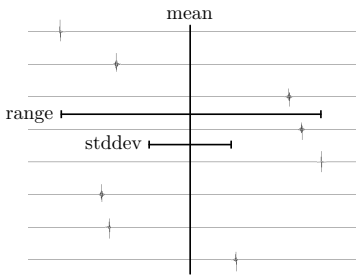

Figure 4: Mean, range, and standard deviation of clicks within a group.

Each time the server triggered a click, we make a group by selecting one click of each device. For each group over time, we measure the total time-span, noted *range* in the figures. We also compute the mean time, the difference of each click from the mean, and their standard deviation, noted *stddev*. (See Figure 4.)

For various reasons, some clicks were not rendered by the devices and these are ignored here. Some other clicks rendered too late and they appear as outliers.

We start by measuring 8 local browser windows (type A), that use the same hardware sound interface as the multi-track recorder, so they all share the same audio clock. In Figure 5, we observe that even during the training stage, the accuracy is better than the audio block duration, mainly because of

Levenshtein distance, which is the sum of added and removed characters.

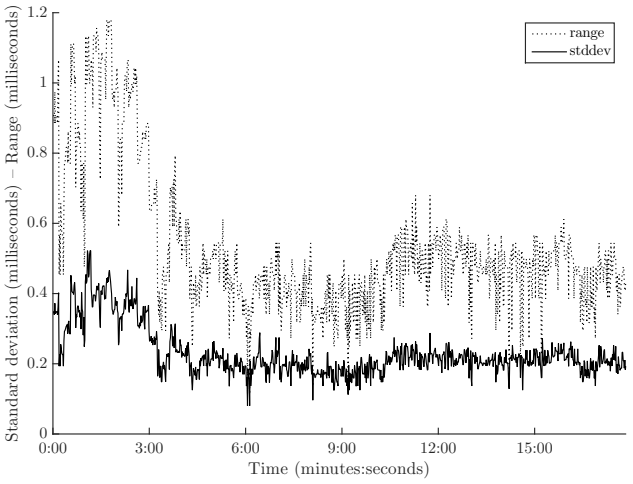

Figure 5: Synchronisation of 8 browser windows on a single desktop computer (type A), also running the server, and the recorder. 11.6 ms audio block.

the mean *offset* over 3 values.[14] After quick stabilisation, the accuracy is always better than 0.8 ms. This confirms that the reference time is precise, and its estimation correct. One of these windows serves as the audio reference process in Section 4.2.

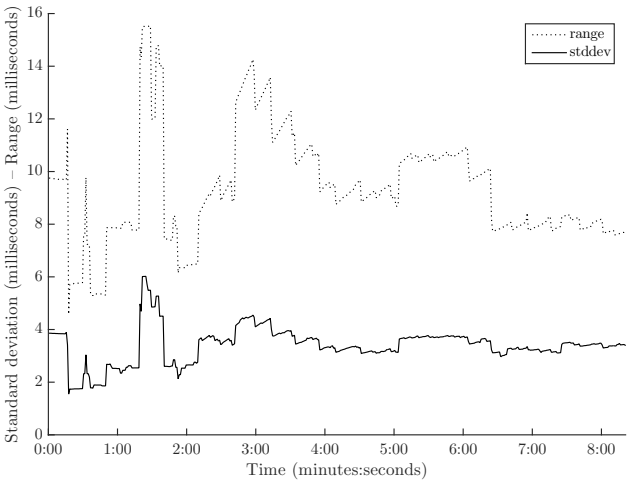

Figure 6: Beginning of synchronisation of 8 devices of type B. 5.8 ms audio block.

Then we measure 8 mobile devices (type B), that use the WLAN, and their own local clock. In Figure 6, we observe that after 2 min the devices start the estimation of the reference clock's frequency. These estimations get better and better over time, as the local drift (seen as the local slope) progressively diminishes. We also note that the local drifts do not imply a loss of accuracy comparing to the training stage. The synchronisation process is stable and the most accurate after 15 min,[15] and the *range* is approximately equals to the

[14] We tried removing it. See Section 3.1.3

[15] See Figure 7 for an equivalent long-term measurement.

audio block duration, while the standard deviation is one-third of it.

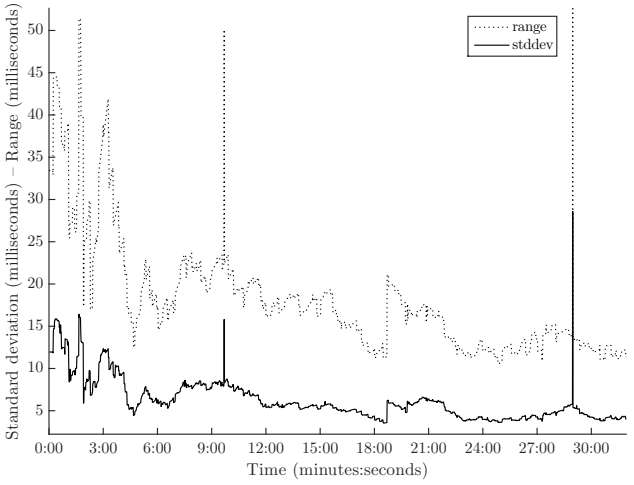

Figure 7: Synchronisation of 8 devices of type C, over long-term. 85 ms audio block.

Then, we measure that the accuracy is not directly bounded by the audio block duration, by measuring 8 devices of type C, with a long block (85 ms). In Figure 7, after stabilisation, the *range* is approximately one-quarter of the audio block duration, while the standard deviation is around one tenth of it. 2 clicks rendered too late, at 10 and 29 min.

On this long-term observation, we note a discontinuity around 19 min: by looking at the individual recordings and after investigation, we found out that there is a discontinuity in the audio rendering, and time. This seems to sporadically happen on *all* of the mobile devices that we studied (one device at a time). While the synchronisation process recovers from it, this is still bad for audio rendering. We did not experience such a problem on desktop or laptop devices.

4.2 Heterogeneous Devices

To measure heterogeneous devices, we use a local browser as a reference (of type A, as in Figure 5), that always stays synchronised to our reference clock. After the synchronisation of the different devices, we calibrate their relative delays, in order to compensate them during the next measurement. All devices of the same type share the same calibration. Then, we restart the synchronisation process to measure it (except for the reference).

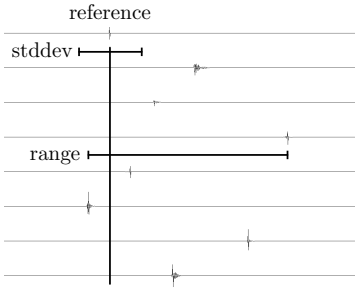

Figure 8: Range of clicks within a group, and standard deviation from reference.

The analysis of the data is the same as before for the *range*, but we measure the difference of each device from the reference, for each click, and we compute the standard deviation from it (see Figure 8).

We mix up devices of type B and of type C, and we measure them with a reference of type A, to check that we obtain the same results as before, with only devices of type C (see Figure 7). Looking at the individual delays from the reference, we confirm that the the overall accuracy is bounded by the less accurate devices, of type C. We note that the different types exhibit different long-term patterns.

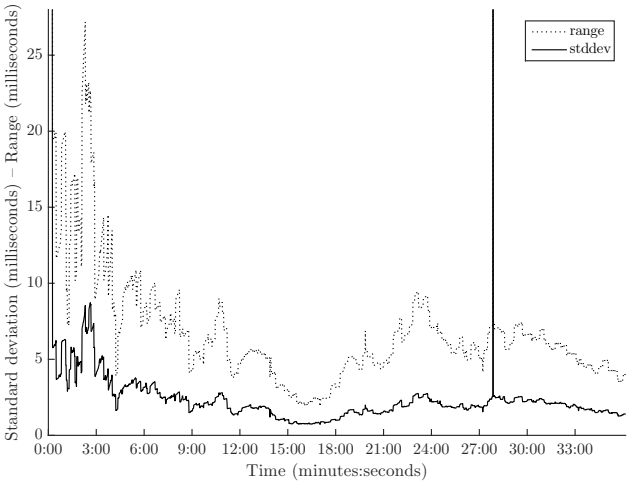

Figure 9: Synchronisation of 7 heterogeneous devices. Standard deviation from reference (type A). 2.9 to 93 ms audio block, 0 to 398 ms calibration.

Then, with the same reference of type A, we measured 7 other devices, of various platforms and characteristics, with an audio block duration ranging from 2.9 to 93 ms. In Figure 9, we observe that during the training the *range* is under 30 ms, while in the long term it oscillates between 5 and 10 ms, with a standard deviation between 1 and 3 ms.

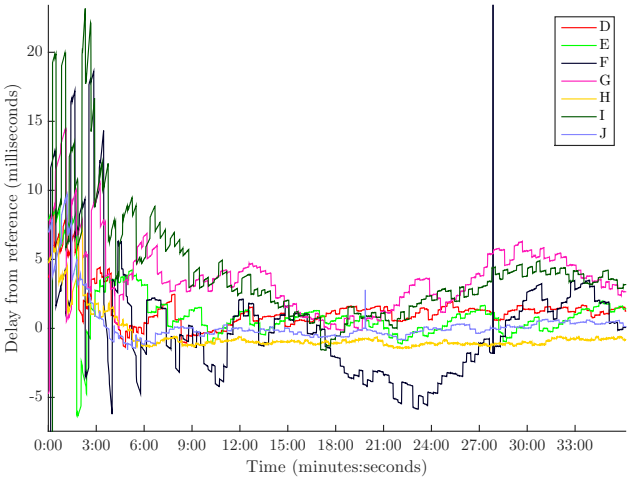

Figure 10: Synchronisation of 7 heterogeneous devices. Individual delays from reference (type A). 2.9 to 93 ms audio block, 0 to 398 ms calibration.

Looking at the individual delays, in Figure 10, we note that the very accurate value around 16 min is an effect of the individual long-term oscillations. While some devices are more accurate, all of them are accurate by 5 ms around the reference after stabilisation. They exhibit various long-term patterns, but all of their local discontinuities is under 2 ms.

5. CONCLUSIONS

We showed that it is already possible to use heterogeneous devices for distributed audio rendering, providing they support standard HTML5 browsers, despite their differences. The requirement on the reference is very light, as it only needs to be able to answer to a ping, so it runs on any standard browser, or server.

We developed a synchronisation solution that is valid for the rendering of audio events, by sharing a reference time, with an individual accuracy of 1 to 10 ms, 20 ms in range, and 0.2 to 5 ms in standard deviation, for all the devices that we measured. This is below their audio block duration.

We successfully used it for a number of realisations, including a distributed loop sequencer, a distributed reverberation processor, and individual instruments sharing a common musical time.

However, some late scheduling may happen, due to network, computation, or unrelated causes on the device. Moreover, we will need to investigate the discontinuities that seem to happen *sometimes* in the Web Audio rendering on all mobile devices.

As the standard is evolving, it will hopefully solve the calibration step by allowing to directly query the latency added after the Web Audio `currentTime`. And the overall accuracy will improve with a native correspondence between `performance.now` and `currentTime`.

6. ACKNOWLEDGEMENTS

We conducted this work in the context of the CoSiMa project, which is supported by the French National Research Agency (`http://cosima.ircam.fr/`, ANR-13-CORD-0010). The authors warmly thank their colleagues Olivier Warusfel, Benjamin Matuszewski, and Samuel Goldszmidt, for their precious contributions, and all the people who let us use their personal device.

7. REFERENCES

[1] J. Blauert. *Spatial Hearing: The Psychophysics of Human Sound Localization*. The MIT Press, revised edition, 1997. 1

[2] A. Deveria. Can I use... Support tables for HTML5, CSS3, etc. `http://caniuse.com/`, 2015. Accessed 2015-10-14. 1

[3] Ecma International. ECMAScript 2015 Language Specification – ECMA-262 6th Edition. `http://www.ecma-international.org/ecma-262/6.0/#sec-time-values-and-time-range`, June 2015. Accessed 2015-09-28. 6

[4] D. Fober, Y. Orlarey, and S. Letz. Real time clock skew estimation over network delays. Technical report, Grame, 2005. 3.1.3

[5] H. W. M. Lunney. Time as heard in speech and music. *Nature*, 249:592, June 1974. 1

[6] J. A. Michon. Studies on subjective duration: I. Differential sensitivity in the perception of repeated temporal intervals. *Acta Psychologica*, 22:441–450, 1964. 1

[7] Mills, Delaware, Martin, Burbank, and Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. Standards Track RFC 5905, Internet Engineering Task Force (IETF), June 2010. 1

[8] D. L. Mills. Executive Summary: Computer Network Time Synchronization. `http://www.eecis.udel.edu/~mills/exec.html`, May 2012. Accessed 2015-02-13. 1

[9] D. L. Mills. Timestamp Capture Principles. `http://www.eecis.udel.edu/~mills/stamp.html`, May 2012. Accessed 2015-02-16. 9

[10] D. L. Mills. Clock Filter Algorithm. `https://www.eecis.udel.edu/~mills/ntp/html/filter.html`, Mar. 2014. Accessed 2015-09-17. 3.1

[11] Multi-device Timing Community Group. Multi-device Timing for Web, Community Group Charter. `https://webtiming.github.io/`, 2015. Accessed 2015-09-29. 1

[12] Node.js Foundation. process Node.js v4.1.1 Manual & Documentation. `https://nodejs.org/api/process.html#process_process_hrtime`, Aug. 2015. Accessed 2015-09-28. 5

[13] NTP.org. Clock Quality. `http://www.ntp.org/ntpfaq/NTP-s-sw-clocks-quality.htm`, Mar. 2014. Accessed 2015-02-13. 11

[14] NTP.org. How does it work? `http://www.ntp.org/ntpfaq/NTP-s-algo.htm`, 2014. Accessed 2015-10-06. 3.1

[15] J. Pätynen. *A Virtual Symphony Orchestra for Studies on Concert Hall Acoustics*. PhD thesis, Aalto University School of Science, Espoo, Finland, Nov. 2011. 1

[16] W3C. Web Audio API. `https://webaudio.github.io/web-audio-api/#widl-AudioContext-currentTime`. Accessed 2015-09-29. 3.1.1

[17] W3C. High Resolution Time. `http://www.w3.org/TR/hr-time/#sec-DOMHighResTimeStamp`, Dec. 2012. Accessed 2015-09-28. 4

[18] W3C. HTML5. `http://www.w3.org/TR/html5/`, Oct. 2014. Accessed 2015-10-14. 1

[19] W3C. Test the Web Forward. `http://testthewebforward.org/`, 2014. Accessed 2015-10-14. 1

[20] W3C Audio WG. Map AudioContext times to DOM timestamps · Issue #340 · WebAudio/web-audio-api · GitHub. `https://github.com/WebAudio/web-audio-api/issues/340`, 2015. Accessed 2015-10-12. 3.2

[21] W3C Audio WG. Need a way to determine AudioContext time of currently audible signal · Issue #12 · WebAudio/web-audio-api · GitHub. `https://github.com/WebAudio/web-audio-api/issues/12`, 2015. Accessed 2015-09-28. 3.1.1

[22] C. Wilson. A Tale of Two Clocks - Scheduling Web Audio with Precision - HTML5 Rocks. `http://www.html5rocks.com/en/tutorials/audio/scheduling/`, Jan. 2013. Accessed 2015-02-02. 3