Media Synchronization on the Web

Ingar M. Arntzen, Njål T. Borch and François Daoust

Abstract The Web is a natural platform for multimedia, with universal reach, powerful backend services and a rich selection of components for capture, interactivity and presentation. In addition, with a strong commitment to modularity, composition and interoperability, the Web should allow advanced media experiences to be constructed by harnessing the combined power of simpler components. Unfortunately, with timed media this may be complicated, as media components require synchronization to provide a consistent experience. This is particularly the case for distributed media experiences. In this chapter we focus on temporal interoperability on the Web, how to allow heterogeneous media components to operate consistently together, synchronized to a common timeline and subject to shared media control. A programming model based on external timing is presented, enabling modularity, interoperability and precise timing among media components, in single-device as well as multi-device media experiences. The model has been proposed within the W3C Multi-device Timing Community Group as a new standard, and this could establish temporal interoperability as one of the foundations of the Web platform.

Key words: media synchronization, media orchestration, motion model, timing object, multi-device

Ingar M. Arntzen
Norut Northern Research Institute, Tromsø, Norway e-mail: ingar.arntzen@norut.no

Njål T. Borch
Norut Northern Research Institute, Tromsø, Norway e-mail: njaal.borch@norut.no

François Daoust
World Wide Web Consortium (W3C), Paris, France e-mail: fd@w3.org

1 Introduction

The Web is all about modularity, composition and interoperability, and this applies across the board, from layout and styling defined in *Hypertext Markup Language (HTML)* to JavaScript-based tools and frameworks. Unfortunately, there is a notable exception to this rule. Composing presentations from multiple, timed media components is far from easy. For example, consider a Web page covering motor sport, using Web Audio [42] for sound effects and visuals made from HTML5 [19] videos, a map with timed GPS-data, WebGL [45] for timed infographics, a timed Twitter [39] widget for social integration, and finally an ad-banner for paid advertisements timed to the race.

In this chapter we focus on media synchronization challenges of this kind, making multiple, heterogeneous media components operate consistently with reference to a common timeline, as well as common media control. We call this *temporal interoperability*. Lack of support for temporal interoperability represents a significant deviation from the core principles of the Web. Harnessing the combined powers of timed media components constitutes a tremendous potential for Web-based media experiences, both in single-device and multi-device scenarios.

The key to temporal interoperability is finding the right approach to media synchronization. There are two basic approaches: *internal timing* or *external timing*. Internal timing is the familiar approach, where media components are coordinated by manipulating their control primitives. External timing is the opposite approach, where media components are explicitly designed to be parts of a bigger experience, by accepting direction from an external timing source.

Though internal timing is currently the predominant approach in Web-based media, external timing is the key to temporal interoperability. If multiple media components are connected to the same source of external timing, synchronized behavior across media components follows by implication. This simplifies media synchronization for application developers. Furthermore, by allowing external timing sources to be synchronized and shared across a network, external timing is also a gateway to precise, distributed multimedia playback and orchestration on the Web platform.

This chapter provides an introduction to external timing as well as the flexible media model and programming model that follow from this approach. The programming model is proposed for standardization within the *W3C Multi-device Timing Community Group (MTCG)* [32] to encourage temporal interoperability on the Web platform. The *timing object* is the central concept in this initiative, defining a common interface to external timing and control for the Web. The MTCG has published a draft specification for the timing object [7] and also maintains *Timingsrc* [3], an open source JavaScript implementation of the timing object programming model. The chapter also describes the *motion model*, which provides online synchronization of timing objects.

The chapter is structured as follows: Section 3 defines media synchronization as a term and briefly presents common challenges for synchronization on the Web. The motion model is presented in Section 4 followed by an introduction to tempo-

ral interoperability and external timing in Section 5. Section 6 surveys the abilities and limitations of Web technologies with respect to media synchronization. Section 7 gives a more detailed presentation of the motion model, including online motion synchronization, synchronization of AV media and timed data. Evaluation is presented in Section 9. Section 10 briefly references the standardization initiative before conclusions are given in Section 11.

2 Central Terms

This section lists central terms used in this chapter.

Timeline: logical axis for media presentation. Values on the timeline are usually associated with a unit, e.g. seconds, milliseconds, frame count or slide number. Timelines may be infinite, or bounded by a range (i.e minimum and maximum values).

Clock: a point moving predictably along a timeline, at a fixed, positive rate. Hardware clocks ultimately depend on a crystal oscillator. System clocks typically count seconds or milliseconds from *epoch* (i.e. 1. Jan 1970 UTC), and may be corrected by clock synchronization protocols (e.g. NTP [27], PTP [14]). From the perspective of application developers, the value of a clock may be read, but not altered.

Motion: a unifying concept for *media playback* and *media control*. Motion represents a point moving predictably along a timeline, with added support for flexibility in movement and interactive control. Motions support discrete jumps on the timeline, as well as a variety of continuous movements expressed through velocity and acceleration. Not moving (i.e. paused) is considered a special case of movement. Motion is a generalization over classical concepts in multimedia, such as *clocks*, *media clocks*, *timers*, *playback controls*, *progress*, etc. Motions are implemented by an *internal clock* and a *vector* describing current movement (position, velocity, acceleration), timestamped relative to the internal clock. Application developers may update the movement vector of a motion at any time.

Timed data: data whose temporal validity is defined in reference to a timeline. For instance, the temporal validity of subtitles are typically expressed in terms of points or intervals on a media *timeline*. Similarly, the temporal validity of video frames essentially maps to frame-length intervals. Timed scripts are a special case of timed data where data represents functions, operations or commands to be executed.

Continuous media: Typically audio or video data. More formally, a subset of *timed data* where media objects cover the timeline without gaps.

Media component: essentially a player for some kind of timed data. Media components are based on two basic types of resources: timed data and motion. The timeline of timed data must be mapped to the timeline of motion. This way, motion defines the temporal validity of timed data. At all times, the media component works

to produce correct media output in the UI, given the current state of timed data and motion. A media component may be anything from a simple text animation in the *Document Object Model (DOM)*, to a highly sophisticated media framework.

User agent: any software that retrieves, renders and facilitates end user interaction with Web content, or whose user interface is implemented using Web technologies.

Browsing context: JavaScript runtime associated with Web document. Browser windows, tabs or *iframes* each have their own browsing context.

Iframe: Web document nested within a Web document, with its own browsing context.

3 Media Synchronization

Dictionary definitions of *media synchronization* typically refer to presentation of multiple instances of media at the same moment in time. Related terms are *media orchestration* and *media timing*, possibly emphasizing more the importance of media control and timed scheduling of capture and playback. In this chapter we use the term *media synchronization* in a broad sense, as a synonym to *media orchestration* and *media timing*. We also limit the definition in a few regards:

- Media synchronization on the Web is client-side and clock-based. The latencies and heterogeneity of the Web environment requires a clock-based approach for acceptable synchronization.
- Media synchronization involves a media component and a clock. The term *relative synchronization* is reserved for comparisons between two or more synchronized media components.

3.1 Challenges

Media synchronization has a wide range of use-cases on the Web, as illustrated by Table 1. Well known use-cases for synchronization within a single Web page include multi-angle video, accessibility features for video, ad-insertion, as well as media experiences spanning different media types, media frameworks, or iframe boundaries. Synchronization across Web pages allow Web pages to present alternative views into a single experience, dividing or duplicating media experiences across devices. Popular use-cases in the home environment involve collaborative viewing, multi-speaker audio, or big screen video synchronized with related content on handheld devices. The last use-cases on the list target global scenarios, such as distributed capture and synchronized Web visualizations for a global audience.

Table 1 Common challenges for media synchronization on the Web.

Synchronization challenges	Use-cases
across media sources	multi-angle video, ad-insertion
across media types	video, WebAudio, animated map
across iframes	video, timed ad-banner
across tabs, browsers, devices	split content, interaction
across platforms	Web, native, broadcast
across people and groups	collaboration, social
across Internet	global media experiences

3.2 Approach

The challenges posed by all these use-cases may be very different in terms of complexity, requirements for precision, scale, infrastructure and more. Still, we argue that a single, common solution would be beneficial. Implementing specific solutions for specific use-cases is very expensive and time-consuming, and lays heavy restrictions on reusability. Even worse, circumstances regarding synchronization may change dynamically during a media session. For instance, a smartphone involved in synchronization over the local network will have to change its approach to media synchronization once the user leaves the house, or switches from WiFi to the mobile network. Crucially though, by solving media synchronization across Internet, all challenges listed above are solved by implication. For instance, if video synchronization is possible across Web pages on the Internet, then synchronizing two videos within the same Web page is just a special case. It follows that the general solution to media synchronization on the Web is distributed and global in nature. Locality may be exploited for synchronization, yet only as optimization.

4 The Motion Model

The primary objectives of the motion model are global synchronization, *Web availability* and simplicity for Web developers. Global synchronization implies media synchronization across the Internet. Web availability means that no additional assumptions can be introduced for media synchronization. If a Web browser is able to load an online Web page, it should also be able to synchronize correctly. The model proposed for this can be outlined in three simple steps:

- Media clock and media controls are encapsulated in one concept, and represented as a stateful resource. This chapter uses the term *motion*[1] for this concept.

[1] *motion* as in *motion pictures*. *Moving through media* still remains a good way to conceptualize media experiences, not least as media experiences become virtual and immersive.

- A *motion* is an online resource, implying that it is hosted by a server and identifiable by a *Universal Resource Locator (URL)*.
- *Media components*[2] synchronize themselves relative to online motions.

According to the model, media synchronization should be a consequence of connecting multiple media components to the same online motion. This way, rich synchronized multi-device presentation may be crafted by connecting relevant media components to the same online motion, as illustrated in Fig. 1.

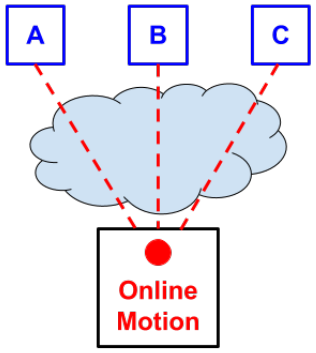

Fig. 1 Media components on three different devices (A,B,C), all connected to an online motion (red circle). Media control requests (e.g. pause/resume) target the online motion and are transmitted across the Internet (light blue cloud). The corresponding state change is communicated back to all connected media components. Each media component adjusts its behaviour independently.

Importantly, the practicality of the motion model depends on Web developers being shielded from the complexities of distributed synchronization. This is achieved by having a *timing object* locally in the Web browser. The timing object acts as an intermediary between media components and online motions, as illustrated by Fig. 2. This way, the challenge of media synchronization is divided in two parts.

- *motion synchronization*: timing object precisely synchronized with online motion (Internet problem).
- *component synchronization*: media component precisely synchronized with timing object (local problem).

Motion synchronization ensures that timing objects connected to the same online motion are kept precisely synchronized. The logic required for motion synchronization could be supported by Web browsers natively (if standardized), or imported into Web pages as a third party JavaScript library. Motion synchronization is outlined in Section 7.3.

[2] *media component:* anything from a simple DOM element with text, to a highly sophisticated media player or multimedia framework.

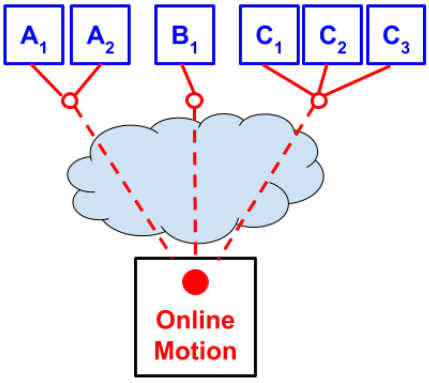

Fig. 2 Timing objects (red unfilled circles) mediate access to online motion. Timing objects may be shared by independent media components within the same browing context.

Component synchronization implies that a media component continuously strives to synchronize its activity relative to a timing object. As such, component synchronization is a local problem. Media components always interface with timing objects through a well defined *Application Programmer Interface (API)* (see Section 7.1). Examples of component synchronization are provided in Section 7.4 and Section 7.5.

5 Temporal Interoperability

Temporal interoperability implies that multiple, possibly heterogeneous media components may easily be combined into a single, consistently timed media experience [5]. We argue that temporal interoperability must be promoted as a principal feature of the Web, and finding the right approach to media synchronization is key to achieving this. In this section we distinguish two basic approaches, *internal timing* and *external timing*, and explain why external timing is better suited as a basis for temporal interoperability. Note that extenal timing is provided by motions[3] according to the motion model.

5.1 Internal timing

Internal timing (Fig. 3 - left) is the most familiar approach, where media components are typically media players or frameworks internalizing aspects of timing and

[3] mediated by timing objects

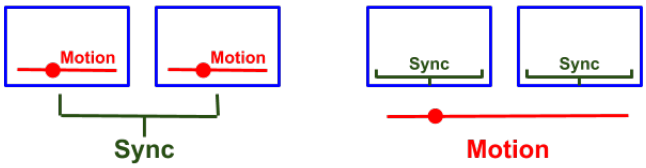

Fig. 3 Blue rectangles represent media components, red symbols represent motion, and green symbols represent the process of media synchronization. To the left: internal timing and external media synchronization. To the right: external timing and internal media synchronization.

control. Synchronizing such media components is an external process, utilizing the control primitives provided by each component.

Internal timing means that media components implement timed operations internally, based on the system clock or some other (hardware) timer and state representing motion. Timed operations may also be affected by other internal factors, such as the buffering of timed data, time consumption in processing, or delays in UI pipelines. The outside world is typically given access to the internal motion of the media component via interactive elements in the *User Interface (UI)*, or programmatically through control primitives defined in the component API. For example, the HTML5 media element allows media playback to be requested by clicking the play button or invoking the play method. The media element then organizes media playback in its own time, subject to delays in initialisation procedures, buffering, decoding and AV-subsystems.

5.2 External timing

External timing (Fig. 3 - right) is the opposite approach, where media components consider themselves parts of a bigger experience. Such media components are explicitly designed to take direction from an external motion, and always do their best to synchronize their own behavior accordingly. If multiple media components are connected to the same external motion, synchronized behavior across media components follows by implication. In this approach, media synchronization is redefined as an internal challenge, to be addressed by each media component independently.

Media control: The external timing approach implies that control over the media component is exercised indirectly, by manipulating the external motion instead of the media component. For instance, if the external motion is paused or time-shifted, the media component must react accordingly. Appropriate controls for the media component may still be exposed through the UI or API of the component. However, such control requests must be routed to the external motion. This ensures that control applies to all media components connected to the same external motion. It also ensures that media components may process control requests without regard to the origin of the request. Media components directed by external motions may still make

use of an internal clock. Importantly though, the external motion takes precedence, so deviations must be compensated for by adjusting the internal clock.

Precision: Precision is a key ambition in media synchronization. With internal timing, synchronization with other media is performed using the control primitives that each media component defines. In the Web environment, such control primitives have typically not been designed with precise timing in mind (see Section 6.1). This makes high quality synchronization hard to achieve. In this model media synchronization generally gets more difficult as the number of components increases. Heterogeneity in media types and control interfaces complicate matters further. For precise synchronization, external timing appears to be a better approach. Media synchronization is solved internally in media components, where it can be implemented with unrestricted access to the internal state and capabilities of the component. Furthermore, the synchronization task is shifted from external application developers to the author of the media component. This makes sense, as the author likely has better understanding of how the media component works. It also ensures that the problem may be solved once, instead of repeatedly by different application developers.

Buffering: Another distinctive feature of the external motion approach is that motion is not sensitive to the internal state (e.g. data availability) of any media component. For instance, external motion might describe playback while a particular media component still lacks data. In the external motion approach, media components must always align themselves with the external motion, to the best of their abilities. For example, media components may adapt by buffering data further ahead, changing to a different data source (e.g. lower bitrate) or even changing to a different presentation mode (e.g audio only). This way, playback may continue undisturbed and media components join in as soon as they are able to. This is particularly important in multi-device scenarios, where a single device with limited bandwidth might otherwise hold back the entire presentation. On the other hand, if the readiness of a particular media component is indeed essential to the experience, this may be solved in application code, by pausing and resuming the external motion.

Master-Slave: Asymmetric master slave synchronization is a common pattern in media synchronization. The pattern implies that internal motion of a master media component is used as external motion for slave media components. However, with multiple media components all but one must be a slave. In the external timing approach all media components are slaves, and the external motion itself is the master. This avoids added complexities of the master-slave pattern, and provides a symmetric model where each media component may request control via the external motion. On the other hand, if asymmetry is indeed appropriate for a given application, this may easily be emulated. For instance, applications may ensure that only one specific media component may issue control requests to the external motion.

Live and on-demand: Solutions for live media often target minimized transport latency for real-time presentation. In other words, the internal motion of live media components is tied to data arrival. This may be problematic in some applications, as differences in transport latency imply that media components will be out of sync.

For example, live Web-based streaming solutions may be seconds apart, even on the same network. Timing issues with live media are even more evident in rich media productions involving multiple live feeds with very different production chains and transport mechanisms. The external timing approach provides the control and flexibility needed for applications to deal with these realities in appropriate ways. With an external motion representing the official live motion, multiple live media sources may be presented in a time-consistent way across components and screens. Such a live motion could be selected to cover at least a majority of viewers. Furthermore, inability to follow the official live motion would be detected by media components internally, potentially triggering application specific reactions. For instance, the viewer could be prompted to switch to a private, slightly time-shifted motion suitable for his or her specific environment.

6 State of the Web

With temporal interoperability established as a goal for the Web, this section surveys current abilities and limitations of the Web with respect to media synchronization. The Web platform[4] is composed of a series of technologies centered around the *Hypertext Markup Language (HTML)*. These technologies have been developed over the years and have grown steadily since the advent of *HTML5* [19], allowing Web applications to access an ever-increasing pool of features such as local storage, geolocation, peer-to-peer communications, notifications, background execution, media capture, and more. This section focuses on Web technologies that produce or consume timed data, and highlights issues that arise when these technologies are used or combined with others for synchronization purposes. These issues are classified and summarized at the end of the section. Please note that this section is written early 2017, and references technologies that are still under development.

6.1 HTML

First versions of the HTML specification (including HTML3.2 [18]) were targeting static documents and did not have any particular support for timed playback. HTML5 introduced the Audio and Video media elements to add support for audio and video data playback. Web applications may control the playback of media elements using commands such as *play* or *pause* as well as properties such as *currentTime* (the current media offset) and *playbackRate* (the playback speed). In theory, this should be enough to harness media element playback to any synchronization logic that authors may be willing to implement. However, there are practical issues:

[4] In this chapter, the Web is seen through the eyes of an end user browsing the Web with his/her favorite *user agent* in 2017.

1. The playback offset of the media element is measured against a media clock, which the specification defines as: *user-agent defined, and may be media resource-dependent, but [which] should approximate the user's wall clock.* In other words, HTML5 does not impose any particular clock for media playback. One second on the wall clock may not correspond to one second of playback, and the relationship between the two may not be linear. Two media elements playing at once on the same page may also follow different clocks, and thus media offset of these two media elements may diverge over time even if playback was initiated at precisely the same time.
2. HTML5 gives no guarantee about the latency that the software and the hardware may introduce when the play button is pressed, and no compensation is done to resorb that time afterwards.
3. The media clock in HTML5 automatically pauses when the user agent needs to fetch more data before it may resume playback. This behavior matches the expectations of authors for most simple media use cases. However, more advanced scenarios where media playback is just a part of a larger and potentially cross-device orchestration would likely require that the media clock keeps ticking no matter what.
4. The *playbackRate* property was motivated by the fast forward and rewind features of *Digital Video Disc (DVD)* players and previously *Videocassette Recorders (VCR)*. It was not meant for precise control of playback velocity on the media timeline.

To address use cases that would require synchronized playback of media elements within a single page, for instance to play a sign language track as an overlay video on top of the video it describes, HTML5 introduced the concept of a *media controller* [23]. Each media element can be associated with a media controller and all the media elements that share the same media controller use the same media clock, allowing synchronized playback. In practice though, browser vendors did not implement media controllers and the feature was dropped in HTML5.1 [21]. It is also worth noting that this mechanism was restricted to media elements and could not be used to orchestrate scenarios that involved other types of timed data.

While sometimes incorrectly viewed as a property of the JavaScript language, the *setTimeout*, *setInterval* and other related timer functions, which allow apps to schedule timeouts, are actually methods of the *window* interface, defined in HTML5. These methods take a timeout counter in milliseconds, but the specification only mandates that Web browsers wait until at least this number of milliseconds have passed (and only provided the Web page has had the focus during that time). In particular, Web browsers may choose to wait a further arbitrary length of time. This allows browsers to optimise power consumption on devices that are in low-power mode. Even if browsers do not wait any further, the event loop may introduce further delays (see Section 6.4). Surprisingly, browsers also fire timers too early on occasion. All in all, the precision of timeouts is not guaranteed on the Web, although experience shows that timeouts are relatively reliable in practice.

6.2 SMIL and Animations

Interestingly, one of the first specifications to have been published as a Web standard after HTML3.2 [18], and as early as 1998, was the *Synchronized Multimedia Integration Language (SMIL)* 1.0 specification [35]. SMIL allowed integrating a set of independent multimedia objects into a synchronized multimedia presentation. SMIL 1.0 was the first Web standard to embed a notion of timeline (although it was only implicitly defined). The specification did not mandate precise synchronization requirements: *the accuracy of synchronization between the children in a parallel group is implementation-dependent.* Support for precise timing has improved in subsequent revisions of SMIL, now in version 3.0 [36].

No matter how close to HTML it may be, SMIL appears to Web application developers as a format on its own. It cannot simply be added to an existing Web application to synchronize some of its components. SMIL has also never been properly supported by browsers, requiring plugins such as RealPlayer [33]. With the disappearance of plugins in Web browsers, authors are left without any simple way to unleash the power of SMIL in their Web applications.

That said, SMIL 1.0 sparked the SMIL Animation specification [37] in 2001, which builds on the SMIL 1.0 timing model to describe an animation framework suitable for integration with *Extensible Markup Language (XML)* documents. SMIL Animation has notably been incorporated in the Scalable Vector Graphics (SVG) 1.0 specification [34], published as a Web standard immediately afterwards. It took many years for SVG to take over Flash [1] and become supported across browsers, with the notable exception of SMIL animations, which *Microsoft* [26] never implemented, and which *Google* [15] now intends to drop in favor of *CSS Animations* and of the *Web Animations specification*.

While still a draft when this book is written, Web Animations [41] appears as a good candidate specification to unite all Web animation frameworks into one, with solid support from *Mozilla* [29], Google and now Microsoft. It introduces the notion of a *global clock*:

> a source of monotonically increasing time values unaffected by adjustments to the system clock. The time values produced by the global clock represent wall-clock milliseconds from an unspecified historical moment.

The specification also defines the notion of a *document timeline* that provides time values tied to the global clock for a particular document. It is easy to relate the global clock of Web Animations with other clocks available to a Web application (e.g. the *High Resolution Time* clock mentioned in Section 6.5). However, the specification acknowledges that the setup of some animations *may incur some setup overhead*, for instance when the user agent delegates the animation to specialized graphics hardware. In other words, the exact start time of an animation cannot be known a priori.

6.3 DOM Events

The ability to use scripting to dynamically access and update the content, structure and style of documents, was developed in parallel to HTML, with *ECMAScript* (commonly known as JavaScript), and the publication of the *Document Object Model (DOM) Level 1* standard in 1998 [11]. This first level did not define any event model for HTML documents, but was quickly followed by *DOM Level 2* [12] and in particular the DOM Level 2 Events standard [13] in 2000. This specification defines: *a platform- and language-neutral interface that gives to programs and scripts a generic event system.*

DOM events feature a *timeStamp* property used to specify the time relative to the epoch at which the event was created. DOM Level 2 Events did not mandate that property on all events. Nowadays, DOM Events, now defined in the DOM4 standard [40], all have a timestamp value, evaluated against the system clock.

The precision of the timestamp value is currently limited to milliseconds, but Google has now switched to using higher resolution timestamps associated with the *high resolution clock* (see Section 6.5). On top of improving the precision down to a few microseconds, this change also means that the *monotonicity* of timestamp values can now be guaranteed. *Monotonicity* means that clock values are never decreasing. This change will hopefully be included in a future revision of the DOM standard and implemented across browsers.

6.4 The Event Loop

On the Web, all activities (including *events, user interactions, scripts, rendering, networking*) are coordinated through the use of an *event loop*[5], composed of a queue of tasks that are run in sequence. For instance, when the user clicks a button, the user agent queues a task on the event loop to dispatch the *click* event onto the document. The user agent cannot interrupt a running task in particular, meaning that, on the Web, all scripts run to completion before further tasks may be processed.

The event loop may explain why a task scheduled to run in 2 seconds from now through a call to the *setTimeout* function may actually run in 2.5 seconds from now, depending on the number of tasks that need to run to completion before this last task may run. In practice, HTML5 has been carefully designed to optimize and prioritize the tasks added to the event loop, and the scheduled task is unlikely to be delayed by much, unless the Web application contains a script that needs to run for a long period of time, which would effectively freeze the event loop.

Starting in 2009, the Web Workers specification [44] was developed to allow Web authors to run scripts in the background, in parallel with the scripts attached

[5] There may be more than one event loop, more than one queue of tasks per event loop, and event loops also have a micro-task queue that helps prioritizing some of the tasks added by HTML algorithms, but this does not change the gist of the comments contained in this section.

to the main document page, and thus without blocking the user interface and the main event loop. Coordination between the main page and its workers uses message passing, which triggers a *message* event on the event loop.

Any synchronization scenario that involves timed data exposed by some script or event logic will de facto be constrained by the event loop. In turn, this probably restricts the maximum level of precision that may be achieved for such scenarios. Roughly speaking, it does not seem possible to achieve less than one millisecond precision on the Web today if the event loop is involved.

6.5 High Resolution Time

In JavaScript, the *Date* class exposes the system clock to Web applications. An instance of this class represents a number of milliseconds since January 1., 1970 UTC. In many cases, this clock is a good enough reference. It has a couple of drawbacks though:

1. The system clock is not monotonic and it is subject to adjustments. There is no guarantee that a further reading of the system clock will yield a greater result than a previous one. Most synchronization scenarios need to rely on the monotonicity of the clock.
2. Sub-millisecond resolution may be needed in some cases, e.g. to compute the frame rate of a script based animation, or to precisely schedule audio cues at the right point in an animation.

As focus on the Web platform shifted away from documents to applications and as the need to improve and measure performance arose, a need for a better clock for the Web that would not have these restrictions emerged. The High Resolution Time specification [16] defines a new clock, *Performance.now()*, that is both guaranteed to be monotonic and accurate to 5 microseconds, unless the user agent cannot achieve that accuracy due to software or hardware constraints. The specification defines the time origin of the clock, which is basically the time when the *browsing context* (i.e. browser Window, tab or iFrame) is first created. The very recent High Resolution Time Level 2 specification [17] aims to expose a similar clock to background workers, and provide a mechanism to relate times between the browsing context and workers.

It seems useful to point out that the 5 microseconds accuracy was not chosen because of hardware limitations. It was rather triggered by privacy concerns as a way to mitigate so called cache attacks, whereby a malicious Web site uses high resolution timing data to fingerprint a particular user. In particular, this sets a hard limit to precision on the Web, that will likely remain stable over time.

6.6 Web Audio API

At about the same time that people started to work on the High Resolution Time specification, Mozilla and Google pushed for the development of an API for processing and synthesizing audio in Web applications. The Web Audio API draft specification [42] is already available across browsers. It builds upon an audio routing graph paradigm where audio nodes are connected to define the audio rendering.

Sample frames exposed by the Web Audio API have a *currentTime* property that represents the position on the Audio timeline, according to the hardware clock of the underlying sound card. As alluded to in the specification, this clock *may not be synchronized with other clocks in the system*. In particular, there is little chance that this clock be synchronized with the High Resolution Time clock, the global clock of Web Animations, or the media clock of a media element.

The group that develops the Web Audio API at W3C investigated technical solutions to overcome these limitations. The API now exposes the relationship between the audio clock and the high resolution clock, coupled with the latency introduced by the software and hardware, so that Web applications may compute the exact times at which a sound will be heard. This is particularly valuable for cross-device audio scenarios, but also allows audio to be output on multiple sound cards at once on a single device.

6.7 Media Capture

W3C started to work on the Media Capture and Streams specification [22] in 2011. This specification defines the notions of *MediaStreamTrack*, which *represents media of a single type that originates from one media source* (typically video produced by a local camera) and of *MediaStream*, which is a group of loosely synchronized MediaStreamTracks. The specification also describes an API to generate MediaStreams and make them available for rendering in a media element in HTML5.

The production of a MediaStreamTrack depends on the underlying hardware and software, which may introduce some latency between the time when the data is detected to the time when it is made available to the Web application. The specification requires user agents to expose the target latency for each track.

The playback of a MediaStream is subject to the same considerations as those raised above when discussing media support in HTML5. The media clock is implementation-dependent in particular. Moreover, a MediaStream is a *live* element and is not seekable. The *currentTime* and *playbackRate* properties of the media element that renders a MediaStream are *read-only* (i.e. media controls do not apply), and thus cannot be adjusted for synchronization[6].

[6] In the future, it may be possible to re-create a seekable stream out of a MediaStream, thanks to the *MediaRecorder* interface defined in the MediaStream Recording specification [25]. This specification is not yet stable when this book is written.

6.8 WebRTC

Work on *Web Real-Time Communication (WebRTC)* and its first specification, the WebRTC 1.0: Real-time Communication Between Browsers specification [46], started at the same time as the work on media capture, in 2011. As the name suggests, the specification allows media and data to be sent to and received from another browser. There is no fixed timing defined, and the goal is to minimize latency. How this is achieved in practice is up to the underlying protocols, which have been designed to reduce latency and allow peer-to-peer communications between devices.

The WebRTC API builds on top of the Media Capture and Streams specification and allows the exchange of MediaStreams. On top of the synchronization restrictions noted above, a remote peer does not have any way to relate the media timeline of the MediaStream it receives with the clock of the local peer that sent it. The WebRTC API does not expose synchronization primitives. This is up to Web applications, which may for instance exchange synchronization parameters over a peer-to-peer data channel. Also, the MediaStreamTracks that compose a MediaStream are essentially treated independently and re-aligned for rendering on the remote peer, when possible. In case of transmission errors or delays, loss of synchronization, e.g. between audio and video tracks, is often preferred in WebRTC scenarios to avoid accumulation of delays and glitches.

6.9 Summary

While the High Resolution Time clock is a step in the right direction, the adoption is still incomplete. As of early 2017, given an arbitrary set of timed data composed of audio/video content, animations, synthesized audio, events, and more there are several issues Web developers need to face to synchronize the presentation:

1. Clocks used by media components or media subsystems may be different and may not follow the system clock. This is typically the case for media elements in HTML5 and for the Web Audio API.
2. The clock used by a media component or a media subsystem may not be monotonic or sufficiently precise.
3. Additionally, specifications may leave some leeway to implementers on the accuracy of timed operations, leading to notable differences in behavior across browsers.
4. Operations may introduce latencies that cannot easily be accounted for. This includes running Web Animations, playing/resuming/capturing media, or scheduling events on the event loop.
5. Standards may require browsers to pause for buffering, as typically happens for media playback in HTML5. This behavior does not play well with the orchestration of video with other types of timed data that do not pause for buffering.

6. The ability to relate clocks is often lost during the transmission of timestamps from one place to another, either because different time origins are used, as happens between an application and its workers, or because the latency of the transmission is not accounted for, e.g. between WebRTC peers. At best, applications developers need to use an out-of-band mechanism to convert timestamps and account for the transport latency.
7. When they exist, controls exposed to harness media components may not be sufficiently fine-grained. For example, the *playbackRate* property of media elements in HTML5 was not designed for precise adjustments, and setting the start time of a Web animation to a specific time value may result in a significant jump between the first and second frames of the animation.

Small improvements to Web technologies should resolve some of these issues, and discussions are underway in relevant standardization groups at W3C when this book is written. For example, timestamps in DOM Events may switch to using the same *Performance.now()* clock. This is all good news for media synchronization, although it may still take time before the situation improves.

We believe that a shift of paradigm is also needed. The Web is all about modularity, composition and interoperability. Temporal aspects have remained an internal issue specific to each technology until now. In the rest of this chapter, a programming model is presented to work around the restrictions mentioned above, allowing media to be precisely orchestrated on the Web, even across devices.

7 Motion

Motion is a simple concept representing playback state (media clock), as well as functions for accessing and manipulating this state (media controls). As such, similar constructs are found in most multimedia frameworks.

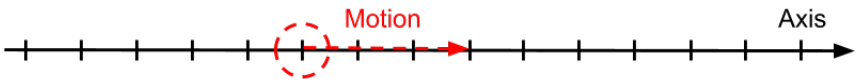

Fig. 4 Motion: point moving along an axis. The current position is marked with a red circle (dashed), and forward velocity of 3 units per second is indicated by the red arrow (dashed).

As illustrated in Fig. 4, motion represents movement (in real-time) of a point, along an axis (timeline). At any moment the point has well defined position, velocity and acceleration[7]. Velocity and acceleration describe continuous movements. Velocity is defined as position-change per second, whereas acceleration is defined

[7] Some animation frameworks support acceleration. Acceleration broadens the utility of motions, yet will likely be ignored in common use cases in classical media (see Section 7.2).

as position- change per second squared. Discrete jumps on the timeline are also supported, simply by modifying the position of the motion. A discrete jump from position A to C implies that the transition took no time, and that no position B (between A and C) was visited. Not moving (i.e. zero velocity and acceleration) is a special case of movement.

Internal State. Motion is defined by an internal clock and a vector (position, velocity, acceleration, timestamp). The vector describes the initial state of the current movement, timestamped relative to the internal clock. This way, future states of the motion may be calculated precisely from the initial vector and elapsed time. Furthermore, application programmers may control the motion simply by supplying a new initial vector. The motion concept was first published under the name Media State Vector (MSV) [6].

7.1 Timing object API

Timing objects provide access to motions. Timing objects may be constructed with a URL to an online motion. If the URL is omitted, it will represent a local motion instead.

```
var URL = "...";
var timingObject = new TimingObject(URL);
```

Listing 1 Constructing a timing object.

The *Timing object API* defines two operations, *query* and *update*, and emits a *change* event as well as a periodic *timeupdate* event.

query(): The query operation returns a vector representing the current state of the motion. This vector includes position, velocity and acceleration, as well as a timestamp. For instance, if a query returns position 4.0 and velocity 1.0 and no acceleration, a new query one second later will return position 5.0.

```
var v = timingObject.query();
console.log("pos:" + v.position);
console.log("vel:" + v.velocity);
console.log("acc:" + v.acceleration);
```

Listing 2 Querying the timing object to get a snapshot vector.

update(vector): The update operation accepts a vector parameter specifying new values for position, velocity and acceleration. This initiates a new movement for the motion. For instance, omitting position implies that the current position will be used. So, an update with velocity 0 pauses the motion at the current position.

```
// play, resume
timingObject.update({ velocity: 1.0 });
// pause
timingObject.update({ velocity: 0.0 });
```

```
// jump and play from 10
timingObject.update({ position: 10.0, velocity: 1.0});
// jump to position 10, keeping the current velocity
timingObject.update({ position: 10.0 });
```

Listing 3 Updating the timing object.

timeupdate event: For compatibility with existing HTML5 media elements and an easy way to update graphical elements, a *timeupdate* evens is emitted periodically.

change event: Whenever a motion is updated, event listeners on the timing object (i.e. media components) will immediately be invoked. Note that the *change* event is not emitted periodically like the *timeupdate* event of HTML5 media elements. The change event signifies the start of a new movement, not the continuation of a movement.

```
timingObject.on("change", function (e) {
  var v = motion.query();
  if (v.velocity === 0.0 && v.acceleration === 0.0) {
    console.log("I'm not moving!");
  } else {
    console.log("I'm moving!");
  }
});
```

Listing 4 Monitoring changes to the motion through the change event.

7.2 Programming with motions

Using motions: Motions are resources used by Web applications, and the developer may define as many as required. What purposes they serve in the application is up to the programmer. If the motion should represent media offset in milliseconds, just set the velocity to 1000 (advances the position of the motion by 1000 milliseconds per second). Or, for certain musical applications it may be practical to let the motion represent beats per second.

Timing converters: A common challenge in media synchronization is that different sources of media content may reference different timelines. For instance, one media stream may have a logical timeline starting with 0, whereas another is timestamped with epoch values. If the relation between these timelines is known (i.e. *relative skew*), it may be practical to create a skewed timing object for one of the media components, connected to the motion. This is supported by *timing converters*. Multiple timing converters may be connected to a motion, each implementing different transformations such as scaling and looping. Timing converters may also be chained. Timing converters implement the *timing object API*, so media components can not distinguish between a timing object and a timing converter. A number of timing converters are implemented in the Timingsrc programming model [3].

Flexibility: The mathematical nature of the motion concept makes it flexible, yet for a particular media component some of this flexibility may be unnecessary, or even unwanted. For instance, the HTML5 media player will typically not be able to operate well with negative velocities, very high velocities, or with acceleration. Fortunately, it does not have to. Instead, the media player may define alternative modes of operation as long as the motion is in an unsupported state. It could show a still image every second for high velocity, or simply stop operation altogether (e.g. black screen with relevant message). Later, when motion re-enters a supported state, normal operation may be resumed for the media player.

7.3 Online Motion

The timing object API is particularly designed to mediate access to online motions, as illustrated by Fig. 5. *Update* operations are forwarded to the online motion, and will not take effect until notification is received from the online motion. After this, a *change event* will be emitted by the timing object. In contrast, *query* is a local (and cheap) operation. This ensures that media components may sample the motion frequently if needed. So, through the Timing Object API, online motions are made available to Web developers as local objects. Only the latency of the update operation should be evidence of a distributed nature.

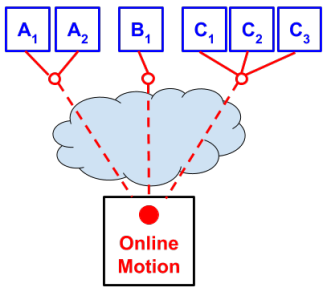

Fig. 5 (Fig. 2 repeated for convenience.) Timing objects (red unfilled circles) mediate access to online motion. Timing objects may be shared by independent media components within the same browsing context.

To support this abstraction, precise, distributed *motion synchronization* is required. In particular, the internal *clock* of the motion must be precisely synchronized with the clock at the online motion server. Synchronized system clocks (e.g. *Network Time Protocol (NTP)* [27] or *Precision Time Protocol (PTP)* [14]) is generally not a valid assumption in the Web domain. As a consequence, an alternative method of estimating a shared clock needs to be used, for example by sampling an online clock directly. In addition, low latency is important for user experiences.

Web agents should be able to join synchronization quickly on page load or after page reload. To achieve this, joining agents must quickly obtain the current vector and the synchronized clock. For some applications the user experience might also benefit from motion updates being disseminated quickly to all agents. Web agents should also be able to join and leave synchronization at any time, or fail, without affecting the synchronization of other agents. Motion synchronization is discussed in more detail in [6].

InMotion is a hosting service for online motions, built by the Motion Corporation [28]. A dedicated online service supporting online motions is likely key to achieving non-functional goals, such as high availability, reliability and scalability. Evaluation of motion synchronization is presented in Section 9.

Finally, the timing object API emphasizes an attractive programming model for multi-device media applications. In particular, by making online motions available under the same API as local motions (see Section 7.1), media components may be used in single-page as well as multi-device media experiences, without modification. Also, by hiding the complexity of distributed motion synchronization, application developers may focus on building great media components using the timing object API. As such, the timing object API provides much needed separation of concern in multi-device media.

7.4 Synchronizing Audio and Video

On the Web, playback of audio and video is supported by HTML5 media elements [20]. Synchronizing media elements relative to a timing object means that the *currentTime* property (i.e. media offset) must be kept equal to the position of the timing object at all times, at least to a good approximation. The basic approach is to monitor the media element continuously, and try to rectify whenever the synchronization error grows beyond a certain threshold. For larger errors *seekTo* is used. This is typically the case on *page load*, or after timing object *change* events. Smaller errors are rectified gradually by manipulating playbackrate. *SeekTo* is quite disruptive to the user experience, so support for variable playbackrate is currently required for high quality synchronization.

MediaSync is a JavaScript library allowing HTML5 media elements to be synchronized by timing objects. The MediaSync library targets usage across the most common Web browsers, so it is not optimized for any particular scenario. Though synchronization of HTML5 media is simple in theory, it involves a few practical challenges, as indicated in Section 6.1. First, *currentTime* is only a coarse representation of the media offset, and it fluctuates considerably when compared to the system clock. The MediaSync library solves this by collecting a backlog of samples, from which a value of currentTime can be estimated. Building up this backlog requires some samples, so it may take more than a second for estimates to stabilize. Another issue relates to unpredictable time-consumption in media control operations. In particular, seekTo(X) will change currentTime to X, but it will require a

non-negligible amount of time to do so. In the context of synchronization, it aims for a fixed target when it should be aiming for a moving target. The MediaSync library compensates for this by overshooting the target. Furthermore, in order to overshoot by the correct amount, the algorithm collects statistics from every seekTo operation. Surprisingly perhaps, this strategy works reasonably well. Evaluation for the MediaSync library is presented in Section 9.

7.5 Synchronizing timed data

Synchronization of timed data using timing objects is an important challenge. Timed data such as subtitles, tracks, scripts, logs or time series typically include items tied to *points* or *intervals* on the timeline. Synchronization then involves activating and deactivating such items at the correct time, in reference to a timing object. To simplify programming of media components based on timed data, a generic *Sequencer* is defined (Fig. 6). The sequencer is similar to the HTML5 track element [38], but is directed by the timing object instead of a HTML5 media element [20]. Web developers register cues associated with intervals on the timeline, and receive event upcalls whenever a cue is activated or deactivated. The sequencer fully supports the timing object, including skipping, reverse playback and acceleration. It may be used for any data type and supports dynamic changes to cues during playback.

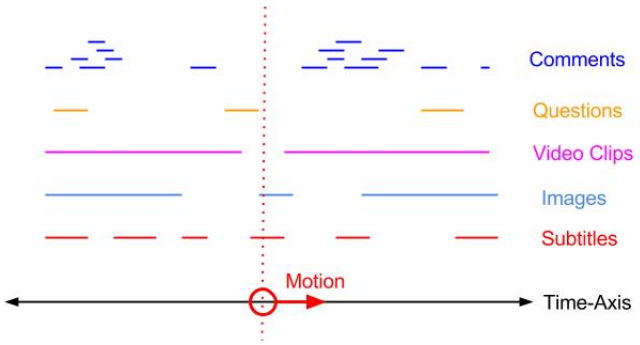

Fig. 6 Sequencing five data sources of timed data, with items tied to intervals on the timeline. Motion along the same timeline defines which items are active (vertical dotted line), and precisely when items will be activated or deactivated.

The sequencer is implemented as a JavaScript library and made available as part of the open-source *Timingsrc* [3] programming model (see Section 10). In the interest of precisely synchronized activation and deactivation and low CPU consumption, the sequencer implementation is not based on frequent polling. Instead, the deterministic nature of the timing object allows events to be calculated and scheduled using *setTimeout*, the timeout mechanism available in Web browsers. Though this

mechanism is not optimized for precision, Web browsers may be precise down to a few milliseconds. The sequencer is presented in further detail in [4].

8 Flexibility and Extensibility

Modern multimedia increasingly demands high flexibility and extensibility. This is driven by a number of strong trends: device proliferation, new sensors, new data types (e.g. sensor data, 3D, 360 degree video), multiple data sources, live data, personalization, interactivity, responsiveness and multi-device support. On top of all this there are also rising expectations to UI design, integration with social networks, and more.

In an attempt to meet such demands, new features have been added to media frameworks allowing programmers to customize the media player to a larger extent. For example, the Flash [1] framework has grown increasingly feature-rich over time, even having partially overlapping features with the Web platform itself. Media Source Extensions (MSE) [24] in HTML5 provide a way to manipulate the video stream client-side. It is also common for media players to expose events and timed cues, allowing custom functionality to be implemented in application code. The text track system of HTML5 is an example of this. MPEG-4 [30] adds support for synchronization and composition of multiple media streams, including timed data such as graphical objects (2D and 3D). In particular, the MPEG-4 Systems part [31] defines an architecture for media clients (terminals) integrating a variety of media formats, delivery methods, interactivity and rendering.

In short, the need for extensibility has driven a development towards standardization of new data formats and features, leading media players to become increasingly sophisticated, yet also more complicated and heavyweight. We call this the *big player* approach to flexibility and extensibility in multimedia.

8.1 Multiple small players

The motion model presents an attractive alternative to the big player approach. The key idea is that a big player may be replaced by multiple smaller players, with precisely synchronized playback. As illustrated in Fig. 7, the flexibility of the motion model allows a variety of specialized media components to be coupled together, forming custom and complex media experiences from simpler parts. We use the term Composite Media [2] for media experiences built in this way.

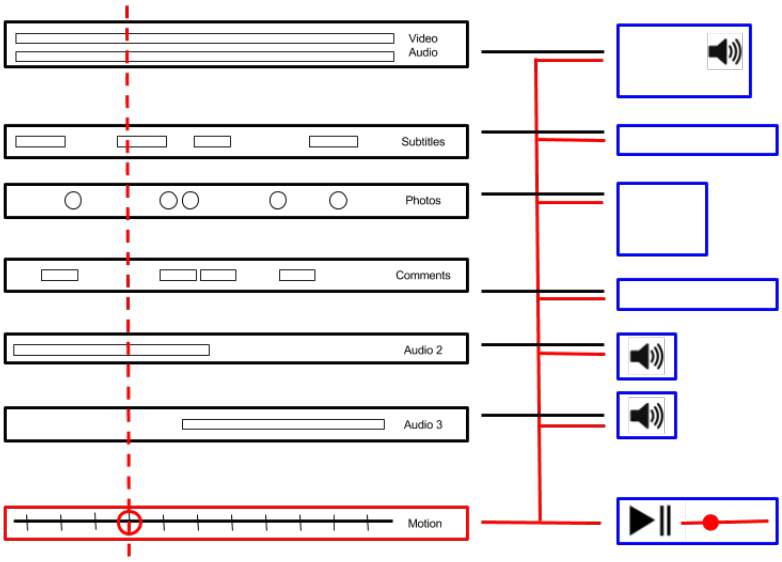

Fig. 7 A single media experience made from multiple media components (blue), possibly distributed across multiple devices. Each media component is connected to motion (red) and a source of timed data (black). There are different types of timed data: an AV container, a subtitle track, photos, comments and two extra audio tracks. The motion defines the timeline for the presentation, and timed data is mapped to this timeline by each media component. Since all the media components are connected to the same motion, they will operate in precise synchrony. One particular media component (bottom media element) provides interactive controls for the presentation, and connects only with motion.

8.2 Dedicated media components

The motion model typically encourages a pattern where each media component is dedicated to solving a small and well defined challenge: Given timed data and a motion, the media component must generate the correct presentation at all times. Such custom media components are implemented in application code, and an appropriate delivery method may be selected for the particular media type and the task at hand. This way, application specific data formats may be integrated into a presentation, as well as standardized formats. Importantly, timed data sources may be dynamic and live, implying that presentations may interact directly with live backend systems and update their presentations during playback.

Media components may also be dedicated with respect to UI. For instance, a single media component may implement interactive controls for the motion, thereby relieving other media components from this added complexity. This encourages a pattern where media components are designed for specific roles in an application, e.g. controllers, viewers and editors, and combined to form the full functionality.

Of course, the fact that these media components are independent may be hidden for end users with appropriate layout and styling, giving the impression of a tightly integrated product. In any case, dedicated media components may be reusable across different views, applications, devices or data sets, as long as APIs to data model and motions remain unchanged.

8.3 Flexible coupling

The motion model allows modularity and flexibility by loose coupling of media components. In fact, media components may be coupled only indirectly through shared motions and shared data sources. This ensures that media components can be added or removed dynamically during playback, or even fail, without disrupting the rest of the presentation. This flexibility is also valuable in development, as media components may be coded and tested in isolation or with other components. New components may always be added without introducing any additional increase in complexity, naturally supporting an incremental development process. Also, the model does not impose restrictions on how motions and timed data sources are connected with media components. A single data source may be shared between multiple media components, or conversely, a single media component may use multiple data sources. The same flexibility goes for motions. There might be multiple aspects of timing and control in an application, requiring multiple motions to be shared between media components.

8.4 Client-side Synthesis

Client-side synthesis is core design principle of the Web platform, and central to key properties such as flexibility, extensibility, reusability and scalability. This principle may now be fully exploited in the context of timed media applications. With the motion model, timed media experiences may be synthesised in real time within the browsing context (client-side), by independent media components working directly on live data sources and motions.

Interestingly, client-side synthesis is not the established approach to linear media, not even in the Web context. With media frameworks such as Flash [1] or MPEG-4 [30], media is typically assembled in a media file or a media container, before being downloaded or streamed to a client-side media player. Essentially, this is server-side synthesis (and client-side playback). While server-side synthesis may have certain advantages (e.g. robustness and simplicity), the disadvantages are also evident. By assembling data within media files and container formats, data is decoupled from its source and effectively flattened into an immutable copy. Introduction of new media types may also be inconvenient, as this must be addressed through standardization of new media and container formats, and support must be implemented

by media players. This may be a time-consuming process. That said, server-side synthesis may still be an appropriate choice for a wide range of media products.

Importantly though, in the motion model the choice between client-side synthesis and server-side synthesis is left to application programmers. Established container-based media frameworks are still usable, provided only that the framework can be integrated and controlled by external motion. Ideally, this integration should be performed internally by the framework. If this is done, frameworks can easily be used in conjunction with native media elements, other frameworks or components that support external motion. If not, integration may also be done externally, subject to the limitations of the framework API. In any case, the motion model relieves media frameworks from the challenge of doing everything, and highlights their value as dedicated, reusable components.

9 Evaluation

The evaluation is concerned with feasibility of the motion model and simplicity for Web developers.

9.1 Motion Synchronization

We have used motion synchronization for a wide range of technical demonstration since 2010. An early evaluation of the research prototype is discussed in the paper titled *The Media State Vector* [6]. Though the interpretations of the experiments are conservative, early findings indicated that motion synchronization could provide frame rate levels of accuracy (33 milliseconds). A few years later, a production ready service called *InMotion* was built by spin off company Motion Corporation [28]. With the introduction of WebSockets [43], results improved significantly. Synchronization errors are in the order of a few milliseconds on all major browsers and most operating systems (including Android). Typically we observe 0-1 millisecond errors for desktop browsers, compared to a system clock synchronized by NTP. The *InMotion* service has also been running continuously for years, supporting a wide range of technical demonstrations, at any time, at any place, and across a wide range of devices. As such, the value of a production grade online service is also confirmed.

Furthermore, the precision of motion synchronization degrades well with poor network conditions. For instance, experiments with video synchronization in *EDGE* connectivity (*Enhanced Data rates for GSM Evolution*) has not been visibly worse, except for longer update latency. In this instance, video data were fetched from local files. Conferences are also notorious hotspots for bad connectivity. In these circumstances, availability of media data consistently fails before synchronization.

9.2 Synchronization of HTML5 media elements

Two technical reports [8, 9] document the abilities and limitations of HTML5 media elements with respect to media synchronization, as well the quality of synchronization achieved by the *MediaSync* library (Fig. 8). Synchronization errors of about 7 milliseconds is reported for both audio and video, on desktops, laptops and high-end smartphones. This corresponds to echoless audio playback. Smartphones and embedded devices such as *ChromeCast* can be expected to provide frame accurate synchronization.

These results have been consistently confirmed by day to day usage over several years. The user experience of multi-device video synchronization is also very good, to the point that errors are hardly visible, as demonstrated by this video [10]. Echoless synchronization with the MediaSync library may also produce various audio effects, like failing to hear one audio source, until volume levels are changed and only the other audio source can be heard. Since these effects are also achieved across browser types and architectures, this is a strong indication that external timing is feasible and already at a useful level.

Synchronization has also been maintained for hours and days at end, without accumulated errors. Loading speeds are also acceptable. Even though the MediaSync library requires about 3 seconds to reach echoless, the experience is perceived as acceptable much before this. A variety of video demonstrations have been published at the Multi-device Timing Community Group Website [32].

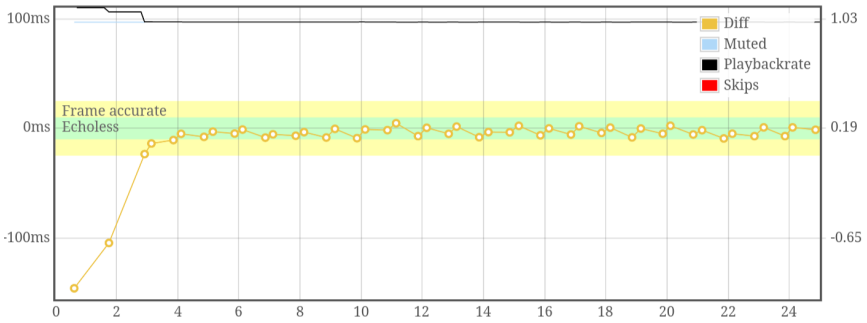

Fig. 8 The figure illustrates an experiment with video (mp4) synchronization on Android using Chrome browser. The plot shows currentTime compared to the ideal playback position defined by motion. The X-axis denotes the timeline of the experiment (seconds). The left Y-axis denotes difference *Diff* (milliseconds) between currentTime and motion. The green band (echoless) is +-10 millisecond and the yellow (frame accurate is) +-25 millisecond. This is achived using variable *playbackRate*. No skips were performed in this experiment. The right Y-axis denotes the value of playbackrate (seconds per second). The media element was muted until playbackrate stabilized.

Though echoless synchronization is generally achievable, a lack of standardization and common tests makes it impossible to provide any guarantees. The experience might also be improved or become broken across software updates. To be able

to support echoless synchronization reliably across browsers and devices, standards must include requirements for synchronization, and testing-suites must be developed to ensure that those requirements are met. Ideally though, media synchronization should be implemented natively in media elements.

9.3 Summary

Interestingly, the results for motion synchronization and HTML5 media synchronization are well aligned with current limitations of the Web platform. For instance, the precision of timed operation in JavaScript is about 1 millisecond, and a 60Hz screen refresh rate corresponds to 16 milliseconds. Furthermore, these results also match limitations in human sensitivity to synchronization errors.

Finally, programming synchronized media experiences in the motion model is both easy and rewarding. In our experience, motions and sequencers are effective thinking tools as well as programming tools. A globally synchronized video experience essentially requires three code statements.

With this, we argue that the feasibility of the motion model is confirmed. It is also clear that synchronization errors in online synchronization are currently dominated by errors in synchronization in HTML5 media elements. Future standardization efforts and optimizations would likely yield significant improvements.

10 Standardization

The Web is widely regarded as a universal multi-media platform although it lacks a common model for timing and media control. The motion model promises to fill this gap, and indicates a significant potential for the Web as a platform for globally synchronized capture and playback of timed multimedia. To bring these possibilities to the attention of the Web community, the motion model has been proposed for Web standardization. The Multi-device Timing Community Group (MTCG) [32] has been created to attract support for this initiative. The MTCG has published the draft specification for the TimingObject [7]. It has also published Timingsrc [3], an open source JavaScript implementation of the TimingObject specification, including timing objects, timing converters, sequencers and the MediaSync library.

Though the ideas promoted by the MTCG have been received with enthusiasm by members within the W3C and within the wider Web community, at present the MTCG proposal has not been evaluated by the W3C.

11 Conclusions

We have explored media synchronization between heterogeneous media components, and highlighted the need for temporal interoperability on the Web platform. While internal timing is the popular approach to Web-based media, external timing is the key to temporal interoperability.

This chapter provided an introduction to external timing as well as the media model and programming model that follow from this approach. By focusing on modularity, loose coupling and client-side synthesis, this media model is well aligned with key design principles of the Web, thereby fully extending the flexibility and extensibility of the Web platform to timed Web applications. Equally important, the external timing approach promises precise distributed playback and media orchestration, enabling precise timing and control also in multi-device Web-based media experiences.

To encourage temporal interoperability on the Web platform, the W3C Multi-device Timing Community Group (MTCG) [32] advocates standardization of the timing object [7] as a common interface to external timing and control. Using the external timing approach, we have demonstrated that the Web is already a strong platform for timed, multi-device media, though it was not designed for it. With standardization it will become even better, likely unleashing a new wave of Web-based creativity across a variety of application domains.

Finally, as the external timing approach to media synchronization works on the Web, it may also be ported to other native applications in the IP environment. This provides a simple mechanism for making distributed media from a mixture of native and Web-based media components.

References

1. Adobe: Adobe Flash. https://www.adobe.com/products/flashruntimes.html
2. Arntzen, I.M., Borch, N.T.: Composite Media, A new paradigm for online media. In: 2013 NEM Summit (Networked Electronic Media), NEM Summit '13, pp. 105–110. Eurescom (2013). URL http://nem-initiative.org/wp-content/uploads/2015/06/2013-NEM-Summit_Proceedings.pdf
3. Arntzen, I.M., Borch, N.T.: Timingsrc: A programming model for timed Web applications, based on the Timing Object. Precise timing, synchronization and control enabled for single-device and multi-device Web applications. http://webtiming.github.io/timingsrc/ (2015)
4. Arntzen, I.M., Borch, N.T.: Data-independent Sequencing with the Timing Object: A JavaScript Sequencer for Single-device and Multi-device Web Media. In: Proceedings of the 7th International Conference on Multimedia Systems, MMSys '16, pp. 24:1–24:10. ACM, New York, NY, USA (2016). DOI 10.1145/2910017.2910614. URL http://doi.acm.org/10.1145/2910017.2910614
5. Arntzen, I.M., Borch, N.T., Daoust, F., Hazael-Massieux, D.: Multi-device Linear Composition on the Web; Enabling Multi-device Linear Media with HTMLTimingobject and Shared Motion. In: Media Synchronization Workshop (MediaSync) in conjunction with ACM TVX 2015. ACM (2015). URL https://sites.google.com/site/mediasynchronization/Paper4_Arntzen_webComposition_CR.pdf
6. Arntzen, I.M., Borch, N.T., Needham, C.P.: The Media State Vector: A unifying concept for Multi-device Media Navigation. In: Proceedings of the 5th Workshop on Mobile Video, MoVid '13, pp. 61–66. ACM, New York, NY, USA (2013). DOI 10.1145/2457413.2457427. URL http://doi.acm.org/10.1145/2457413.2457427
7. Arntzen, I.M., Daoust, F., Borch, N.T.: Timing Object; Draft community group report. http://webtiming.github.io/timingobject/ (2015)
8. Borch, N.T., Arntzen, I.M.: Distributed Synchronization of HTML5 Media. Tech. Rep. 15, Norut Northern Research Institute (2014)
9. Borch, N.T., Arntzen, I.M.: Mediasync Report 2015: Evaluating timed playback of HTML5 Media. Tech. Rep. 28, Norut Northern Research Institute (2015)
10. Video synchronization by Motion Corporation. https://youtu.be/lfoUstnusIE (2015)
11. Document Object Model (DOM) Level-1. https://www.w3.org/TR/REC-DOM-Level-1/ (1998)
12. Document Object Model (DOM) Level-2. https://www.w3.org/TR/DOM-Level-2/ (2000)
13. Document Object Model (DOM) Level-2 Events. https://www.w3.org/TR/DOM-Level-2-Events/ (2000)
14. Eidson, J., Lee, K.: Ieee 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In: Sensors for Industry Conference, 2002. 2nd ISA/IEEE, pp. 98–105. Ieee (2002)
15. Google. https://www.google.com (2017)
16. High Resolution Time. https://www.w3.org/TR/hr-time-1/ (2012)
17. High Resolution Time Level 2. https://www.w3.org/TR/hr-time-2/ (2016)
18. HTML 3.2 Reference Specification. https://www.w3.org/TR/REC-html32 (1997)
19. HTML5. https://www.w3.org/TR/html5/ (2014)
20. HTML5 Media Elements. http://dev.w3.org/html5/spec-preview/media-elements.html (2012)
21. HTML5.1. https://www.w3.org/TR/html51/ (2016)
22. Media Capture and Streams. https://www.w3.org/TR/mediacapture-streams/ (2016)
23. HTML5 Media Controller. https://dev.w3.org/html5/spec-preview/media-elements.html (2014)
24. Media Source Extensions. https://www.w3.org/TR/media-source/ (2016)
25. MediaStream Recording. https://www.w3.org/TR/mediastream-recording/ (2017)
26. Microsoft. https://www.microsoft.com/ (2017)
27. Mills, D.L.: Internet time synchronization: the network time protocol. IEEE Transactions on Communications **39**(10), 1482–1493 (1991). DOI 10.1109/26.103043

28. Motion Corporation. http://motioncorporation.com
29. Mozilla. https://www.mozilla.org (2017)
30. MPEG-4. http://mpeg.chiariglione.org/standards/mpeg-4
31. MPEG-4 Systems. http://mpeg.chiariglione.org/standards/mpeg-4/systems (2005)
32. Multi-device Timing Community Group. https://www.w3.org/community/webtiming/ (2015)
33. RealPlayer. http://www.real.com/ (2017)
34. Scalable Vector Graphics (SVG) 1.1. https://www.w3.org/TR/SVG/ (2011)
35. Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. https://www.w3.org/TR/1998/REC-smil-19980615/ (1998)
36. SMIL 3.0 Synchronized Multimedia Integration Language. http://www.w3.org/TR/REC-smil/ (2008)
37. Smil Animation. https://www.w3.org/TR/smil-animation/ (2001)
38. HTML5 Text Track. http://dev.w3.org/html5/spec-preview/media-elements.html#text-track (2012)
39. Twitter. https://twitter.com/ (2017)
40. W3C DOM4. https://www.w3.org/TR/dom/ (2015)
41. Web Animations. http://www.w3.org/TR/web-animations/ (2016)
42. Web Audio API. https://www.w3.org/TR/webaudio/ (2015)
43. The Web Socket Protocol. https://tools.ietf.org/html/rfc6455 (2011)
44. Web Workers. https://www.w3.org/TR/workers/ (2015)
45. WebGL. https://www.khronos.org/webgl/ (2017)
46. WebRTC Real-time Communication Between Browsers. https://www.w3.org/TR/webrtc/ (2017)