Latest advances in the development of the open-source player dash.js

Daniel Silhavy
Fraunhofer FOKUS
Berlin, Germany
daniel.silhavy@fokus.fraunhofer.de

Stefan Pham
Fraunhofer FOKUS
Berlin, Germany
stefan.pham@fokus.fraunhofer.de

Stefan Arbanowski
Fraunhofer FOKUS
Berlin, Germany
stefan.arbanowski@fokus.fraunhofer.de

Stephan Steglich
Fraunhofer FOKUS
Berlin, Germany
stefan.arbanowski@fokus.fraunhofer.de

Björn Harrer
Deutsche Telekom AG
Darmstadt, Germany
bjoern.harrer@telekom.de

ABSTRACT

The trend to consume high-quality videos over the internet lead to a high demand for sophisticated and robust video player implementations. dash.js is a prominent option for implementing production grade DASH-based applications and products, and is also widely used for academic research purposes. In this paper, we introduce the latest additions and improvements to dash.js. We focus on various features and use cases such as player performance and robustness, low latency streaming, metric reporting and digital rights management. The features and improvements introduced in this paper provide great benefits not only for media streaming clients, but also for the server-side components involved in the media stream process.

CCS CONCEPTS

• **Information systems** → **Multimedia streaming**; • **Security and privacy** → *Digital rights management*.

KEYWORDS

Adaptive Bitrate Streaming, MPEG-DASH, Open-Source Software, dash.js

ACM Reference Format:
Daniel Silhavy, Stefan Pham, Stefan Arbanowski, Stephan Steglich, and Björn Harrer. 2022. Latest advances in the development of the open-source player dash.js. In *Mile-High Video Conference (MHV '22), March 1–3, 2022, Denver, CO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3510450.3517311

1 INTRODUCTION

In 2019, 60% of the global application internet traffic was caused by video streaming applications. [14] Streaming providers such as Netflix and YouTube accounted for 12.6% and 8.7% of the global application traffic, respectively. [14] The two main streaming formats used for the delivery of video content over the internet are HTTP Live Streaming (HLS) and Dynamic Adaptive Streaming over HTTP (MPEG-DASH). According to the Bitmovin Video Developer Report of 2021, HLS is used by 73% of the streaming providers while MPEG-DASH reaches 64% adaption. [4]

The trend to consume high-quality videos over the internet leads to a high demand for sophisticated and robust video player implementations. Unreliable and erroneous player behavior results in bad user experience and potential loss of customers. A major challenge for player development is the heterogeneity of target platforms and devices, ranging from gaming consoles to Smart TVs, mobile devices, and desktop browsers. To benefit from continuous code contributions and improvements coming from the community and to support a wide variety of these platforms and devices, 57% of streaming providers use an open source code-base for their video players. [4]

In this paper, we introduce the latest additions and improvements to **dash.js**, a free, open-source MPEG-DASH player. dash.js is a prominent option for implementing commercial, production grade DASH-based applications and products. Moreover, dash.js serves as a reference client for academic purposes.

The remainder of the paper is structured as follows: Section 2, introduces dash.js and the underlying technology stack. Section 3 covers performance and robustness improvements to dash.js. This includes MPD patching as a mean to reduce the bandwidth and parsing requirements on the client side, as well as the latest improvements in terms of multiperiod playback. Moreover, the reason for gaps in the media buffer and their handling in dash.js is described. Furthermore, ways of recovering from MSE errors are introduced. Section 3 closes with an introduction of the UTC synchronization logic in dash.js. Section 4 discusses the problems that media players are facing when playing with low latency. Low on Latency+ and Learn2Adapt as two new ABR algorithms specifically designed for low latency streaming are introduced. Next, Section 5 covers the Common Media Client Data (CMCD) implementation in dash.js and the benefits for media players and CDNs when using CMCD. The paper closes with a discussion of the latest improvements regarding Digital Rights Management (DRM) in Section 6 and the conclusion in Section 7

MHV '22, March 1–3, 2022, Denver, CO, USA

ACM ISBN 978-1-4503-9222-8/22/03…$15.00
https://doi.org/10.1145/3510450.3517311

2 DASH.JS

dash.js, is a free, open-source MPEG-DASH player. It is available on GitHub [1] and NPM [2] under the BSD license. dash.js is an outcome of the Dash Industry Forum (DASH-IF) , a group of leading streaming companies promoting and catalyzing the adoption of MPEG-DASH. [10] Since 2019 Fraunhofer FOKUS is the official maintainer of the dash.js project and has reviewed and contributed major new features and improvements to the dash.js source-code. [9] dash.js is written in JavaScript and uses the Media Source Extensions (MSE) and the Encrypted Media Extensions Encrypted Media Extensions (EME) defined by World Wide Web Consortium (W3C).

As most of the common target platforms support both MSE and EME, dash.js is a prominent option for a production grade player. A single player, running on all the required target devices and platforms, eliminates the need for maintaining different codebases. This results in time and resource savings. In addition, users of dash.js benefit from the continuous active development of the player and code contributions from the development community. Examples of companies using dash.js in production are British Broadcasting Corporation (BBC) [1] and Deutsche Telekom [3]. dash.js is also widely used in the academic field, for instance for researching new ABR algorithms.

dash.js offers a wide set of features related to adaptive media streaming. This includes playback of dynamic and static content using the DASH and Smooth Streaming format. Moreover, multiple DRM systems (Widevine, Playready, Clearkey) and multiple versions of the EME (0.1b,2014,2015+) are supported. dash.js offers a flexible and extendable buffer and throughput-based ABR decision logic and support for CMAF based low-latency streaming. Furthermore, multiple subtitles formats such as embedded (CEA-608, CEA-708), fragmented (TTML, IMSC1, EBU-TT) and sideloaded (WebVTT, TTML) are supported. Additionally, dash.js offers support for multi-audio, thumbnails and offline playback.

3 PERFORMANCE AND ROBUSTNESS IMPROVEMENTS

This Section covers changes and additions to dash.js regarding player performance and player robustness. First, MPD patching as a mechanism to minimize the amount of data required to update the Media Presentation Description (MPD) on the client side is introduced. Next, improvements for multiperiod playback and gap handling are discussed. In addition, the handling of MSE errors is explained. The Section closes with an illustration of the improved UTC synchronization mechanism in dash.js

3.1 MPD patching

In a typical dynamic streaming session, the Media Presentation Description (MPD) is frequently updated. MPD updates mainly include the addition and removal of media segments and period elements. Although some parts of the MPD can change between two consecutive MPD updates, most parts of it remain unchanged. Consequently, only the clients freshly joining the streaming session need all information from the MPD. For existing clients, a lot of the information in the MPD is already known and does not need to be signaled again.

The size of MPDs using a *<SegmentTimeline>* or *<SegmentList>* mechanism tend to increase very fast. In *Performance Considerations of HTML5-Based Dynamic Packaging for Media Streaming* [16] we examined the relationship between manifest sizes and parsing times of HTML5 based video players on different platforms. We found that parsing of a manifest with a size of 191kb can take up to 2500ms on devices with low computational power. Consequently, the reduction of manifest sizes has a significant influence on the performance of the player and the quality of experience for the end user. In the worst case, a player cannot keep up with the signaled manifest update frequency, simply because the manifest parsing time exceeds the specified update frequency. Another essential point is the bandwidth requirement on the server and the client side. Large manifest files with lots of redundant information lead to wasted bandwidth on the client side and higher delivery costs on the server side.

MPEG-DASH 5th Edition introduces MPD patching as a mechanism to provide only mandatory MPD information to the client, minimizing bandwidth and parsing requirements. The complete MPD is only fetched once at playback start. Subsequent updates to the MPD are provided through MPD patches. MPD patches only contain new information, such as additional media segments. First results from Hulu showed that MPD patching significantly reduce the size and the parsing time of the transferred XML data [5].

dash.js introduced support for MPD patching in version 3.2.1 [4]. The basic workflow for MPD patching in dash.js is depicted in Figure 1. At playback start, the full MPD is requested. For every other MPD update, dash.js checks if a *<PatchLocation>* element is defined on MPD level. The *<PatchLocation>* points to the server that provides the patching information. In case a valid *<PatchLocation>* is defined the patching information are requested, otherwise the full MPD is fetched. MPD patching uses XPath [5] to address specific paths of the XML based manifest file. An example of an MPD patching manifest is depicted in Listing 1. In this example, a new *<S>* element is added to the existing *<SegmentTimeline>* element. Moreover, the *MPD@publishTime* and the *MPD@PatchLocation* are updated. In case the provided MPD patch is invalid or incompatible, the full MPD is requested again. Patches are considered incompatible if either the *MPD@id* does not match the *Patch@mpdId*, or *MPD@publishtime* does not match *Patch@originalPublishTime*, or *MPD@publishtime* is greater than *Patch@publishTime*.

Listing 1: Example of a patch manifest

```
<?xml version="1.0" encoding="UTF-8"?>
<Patch mpdId="channel" originalPublishTime="2019-10-18T22:06:14" publishTime="
       2019-10-18T22:06:17"
       xmlns="urn:mpeg:dash:schema:mpd-patch:2020" xmlns:p="urn:ietf:params:xml:
            schema:patch-ops">
    <p:replace sel="/MPD/@publishTime">2019-10-18T22:06:17 </p:replace>
    <p:replace sel="/MPD/PatchLocation[0]">
        <PatchLocation ttl="60">/patch/channel.mpd?publishTime=2019-10-18T22
            :06:17&s=5095823234</PatchLocation>
    </p:replace>
    <p:add sel="/MPD/Period[@id='1']/AdaptationSet[@id='2']/SegmentTemplate/
        SegmentTimeline">
        <S t="5095823234" d="360360"/>
    </p:add>
</Patch>
```

[1] https://github.com/Dash-Industry-Forum/dash.js
[2] https://www.npmjs.com/dashjs
[3] https://web.magentatv.de/
[4] https://github.com/Dash-Industry-Forum/dash.js/releases/tag/v3.2.1
[5] https://www.w3.org/TR/1999/REC-xpath-19991116/

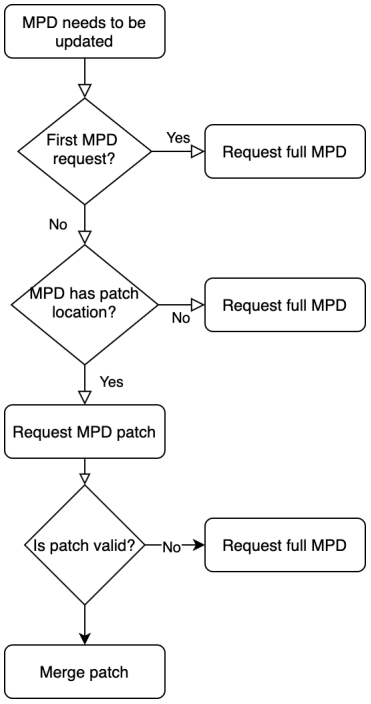

Figure 1: The basic workflow for MPD patching.

3.2 Multiperiod playback

Multiperiod playback is the foundation for important use cases such as ad-insertion and transition between encrypted and non-encrypted content. The structure of a typical multiperiod MPD is depicted in Figure 2. In this case, the main content is interrupted by two ads and a short ad-slate. Ad slates are used in case the total duration of the ads returned by the ad server does not match the duration of the ad opportunity.

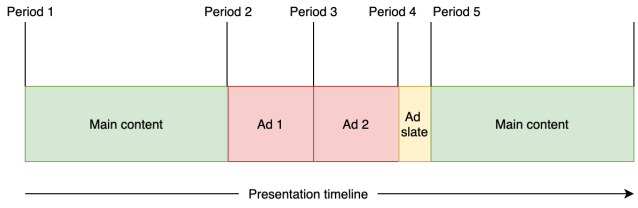

Figure 2: An example of a multiperiod MPD. The main content is interrupted by two ads and a short ad-slate.

dash.js version 4.0 [6] introduced various improvements related to playback of multiperiod content.

One of the new key features is the support for prebuffering multiple upcoming periods. In previous versions of dash.js the prebuffering was limited to a single period. This lead to small buffers and stability problems in case of upcoming short periods such as ad-slates. By prebuffering multiple upcoming periods dash.js maintains a stable buffer level and is less prone to bandwidth fluctuations.

[6] https://github.com/Dash-Industry-Forum/dash.js/releases/tag/v4.0.0

Another important change is the adjustment of the DVR window. Before dash.js 4.0 the DVR window was bound to the currently active period as illustrated in Figure 3. Consequently, seeking was limited to a single period, even if the DVR window defined in the MPD included multiple periods. With dash.js 4.0 the DVR window for multiperiod streams is implemented in a specification-compliant manner, enabling seeking between the periods.

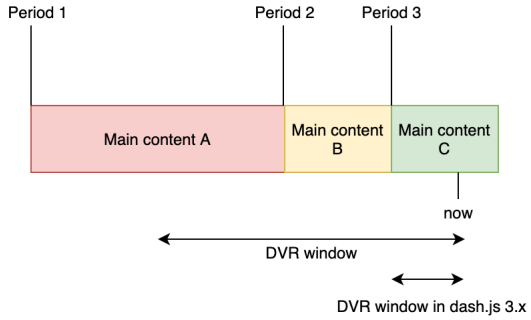

Figure 3: The DVR window in dash.js 3.x is limited to the current active period. dash.js 4.0 removes this restriction.

Another important change in dash.js 4.0 is the support for transition between non-encrypted and encrypted periods. Most MSE based devices require a reset of the MediaSource when switching from non-encrypted to encrypted media segments. This is due to the reason that the SourceBuffers were initialized with an unencrypted initialization segment. dash.js accounts for this requirement and thereby enables playback of streams that start with an unencrypted period and then transition to an encrypted period.

3.3 Gap handling

A huge problem for MSE based players are gaps in the presentation timeline. Most MSE implementations cannot handle situations in which the media buffer is not continuous. They stall as soon as the play position reaches a gap. Figure 4 depicts an example of such a situation. In this case, segment 1 and segment 2 are perfectly aligned, while there is a gap between segment 2 and segment 3. Playback stalls at the end of segment 2 unless the situation is solved by the player.

Figure 4: Gaps in the presentation timeline can lead to playback stalling if not handled by the player.

There are multiple reasons for gaps in the media buffer. For starters, subsequent periods might not align timing wise and consequently segments at period boundaries do not align as well. Another reason are periods with a positive *@eptDelta* or a negative *@pdDelta* as depicted in Figure 5. A positive *@eptDelta* means that the earliest presentation time of the first segment in the period is larger

than the *@presentationTimeOffset*. Consequently, the segment is not aligned with the start of the period. Similar, a negative *@pdDelta* results in a gap at the end of the Period.

Figure 5: A positive *@eptDelta* and negative *@pdDelta* lead to gaps at the start and the end of a period.

Another prominent reason for gaps are video segments that are partially shifted out of a period. MSE based players typically define an append window matching the period boundaries. An append window "represents a single continuous time range with a single start time and end time. Coded frames with presentation timestamp within this range are allowed to be appended to the SourceBuffer while coded frames outside this range are filtered out. The append window start and end times are controlled by the appendWindowStart and appendWindowEnd attributes respectively". [17]

An example of a shifted segment resulting in a negative *@eptDelta* is depicted in Figure 6. By moving segment 1 out of the period all frames up to *Period@start* are outside the append window and are filtered out. Typically, a media segment only contains a single keyframe right at the start of the segment. In this example, the keyframe at the beginning of segment 1 is also filtered out. As a consequence, all media samples from *Period@start* up to the next keyframe can not be decoded and are discarded. This usually means that all samples of segment 1 are filtered (since there is only one I-frame at the beginning), leading to a large gap ranging from *Period@start* to segment 2.

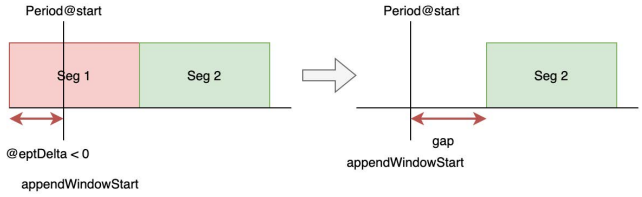

Figure 6: A negative *@eptDelta* shifts a segment partially out of its period. All media samples up to the next keyframe that is included in the period can not be decoded and are discarded

dash.js implements a separate controller to account for possible gaps in the media buffer. Since native implementations stall at gap boundaries, dash.js triggers a seek to the next playable position in the buffer. For static content, the seek can be done immediately. For dynamic content, the player needs to maintain a consistent live edge. For that reason, it needs to wait for the duration of the gap before triggering a seek to the next range in the buffer.

With dash.js version 4.2.0 the existing gap handling implementation was extended to also account for gaps signaled in the MPD. As an example, consider Figure 4 again. The gap the between segment 2 and segment 3 can originate from misaligned segments, but can also be caused by missing segment information in the manifest. Simply put, there might not be a media segment specified in the MPD for the time between segment 2 and segment 3. To handle such scenarios' dash.js will automatically adjust the seek position once a client seeks into the gap. Depending on which segment is closest to the target seek position, either the last part of segment 2, or the first part of segment 3 is fetched and played.

3.4 MSE errors

Errors thrown directly by the MSE typical lead to playback failure. In most cases, an error puts the MSE in a state that it cannot recover from. Hence, manual error recovery by the player is required to deal with such situations. With dash.js version 4.1.0 [7] support for MSE error recovery was introduced. Errors thrown by the MSE are caught and handled by the internal classes responsible for the specific media type (audio or video) that caused the error. For that reason, the current playback position is saved, the segment that caused the error is blacklisted and the MSE is completely reset with all the attached SourceBuffers. After the reset, the player resumes from the previous playback position and omits the blacklisted segment. The resulting gap in the media buffer caused by the blacklisted segment is handled by the gap handler described in Section 3.3

3.5 UTC synchronization

During playback of dynamic presentations, a wall clock is used as the timing reference for player decisions. It is critical to synchronize the clocks of the DASH player and the server components when using a dynamic presentation. The MPD timeline of a dynamic presentation is mapped to wall clock time, and many playback decisions are clock-driven and assume a common understanding of time by the DASH client and server components. [11] [12]

Clock synchronization mechanisms are described by UTCTiming elements in the MPD. [12] With version 4.1.0 dash.js introduced optimization regarding clock synchronization. By default, dash.js performs a clock synchronization at playback start, issuing a request to the timing server. The offset between the client and the server clock is calculated and used for all timing related calculations and decisions. In addition to the single initial request, dash.js performs a predefined number of background requests to verify the initially calculated offset. That way, player startup is not delayed, but the calculated offset is verified.

Moreover, dash.js initiates a synchronization request after each MPD update. This behavior is adjusted dynamically during runtime, as illustrated in Listings 2. $shouldPerformSynchronization()$ compares the current wallclock time against the time of the last sync attempt. In case the difference is larger than $timeBetweenSyncAttempts$ a synchronization request is issued. Otherwise, playback continues without a clock sync.

[7] https://github.com/Dash-Industry-Forum/dash.js/releases/tag/v4.1.0

Listing 2: dash.js code to check if a synchronization request is to be made

```
function shouldPerformSynchronization() {
    try {
        const timeBetweenSyncAttempts = !isNaN(internalTimeBetweenSyncAttempts)
                ? internalTimeBetweenSyncAttempts :
                DEFAULT_TIME_BETWEEN_SYNC_ATTEMPTS;

        if (!timeOfLastSync || !timeBetweenSyncAttempts || isNaN(
                timeBetweenSyncAttempts)) {
            return true;
        }

        return ((Date.now() - timeOfLastSync) / 1000) >= timeBetweenSyncAttempts
                ;
    } catch (e) {
        return true;
    }
}
```

After each regular synchronization attempt, dash.js adjusts its internal *internalTimeBetweenSyncAttempts* parameter based on predefined thresholds. The concrete implementation is depicted in Listing 3. If the difference between the previous and the current client-server offset is within an allowed threshold, then the time to the next sync attempt is increased. Otherwise, the time to the next sync attempt is decreased. The relevant adjustment and configurations parameters can be set as part of the dash.js settings.

Listing 3: dash.js code to adjust the time between UTC syncs

```
function adjustTimeBetweenSyncAttempts(offset) {
     const isOffsetDriftWithinThreshold = _isOffsetDriftWithinThreshold(offset);

    if (isOffsetDriftWithinThreshold) {
        // The drift between the current offset and the last offset is within
                the allowed threshold. Increase sync time
        adjustedTimeBetweenSyncAttempts = Math.min(timeBetweenSyncAttempts *
                timeBetweenSyncAttemptsAdjustmentFactor,
                maximumTimeBetweenSyncAttempts);
    } else {
        // Drift between the current offset and the last offset is not within
                the allowed threshold. Decrease sync time
        adjustedTimeBetweenSyncAttempts = Math.max(timeBetweenSyncAttempts /
                timeBetweenSyncAttemptsAdjustmentFactor,
                minimumTimeBetweenSyncAttempts);
    }

    internalTimeBetweenSyncAttempts = adjustedTimeBetweenSyncAttempts;
}
```

4 LOW-LATENCY STREAMING

One of the major challenges in OTT streaming is reducing the live-streaming latency. This can be crucial for live events like sport games or for an optimal streamer-user interaction in eSports games. The Common Media Application Format (CMAF) introduces the concept of "chunks". A CMAF chunk allows the client to access parts of the segments media data before the segment is completely finished. Compared to "classic" non-chunked streaming, a player for low latency streaming has to overcome additional challenges regarding throughput estimation and maintaining a consistent live edge. In "Performance of Low-Latency HTTP-based Streaming Players", Zhang et al. evaluated six different low latency players and concluded that LL-DASH players can maintain latencies in the range of 3-4 seconds for different network conditions [18].

Common throughput-based ABR algorithms calculate the throughput by dividing the segment size by the download time. This way of throughput calculation is not applicable for clients operating in low latency mode. Since segments are transferred via HTTP 1.1 chunked transfer encoding, the download time of a segment is similar to its duration in case the download of that segment is started prior to its completion. For instance, the download time of a segment with six second duration will be approximately six seconds. There will be idle times in which no data is transferred from the server to the client. However, the connection remains open while the client waits for new data. The total download time includes these idle times. Consequently, the total download time is not a good indicator for the available bandwidth on the client side.

When playing in low latency mode the client needs to maintain a consistent live edge allowing only small deviations compared to the target latency. To maintain a consistent live edge, players typically adjusts the playback rate of the video (catchup mechanism), or perform a seek back to the live edge.

dash.js version 3.2.0 introduced two low latency-specific ABR algorithms, namely Low on Latency+ (LoL+) [3] and Learn2Adapt (L2A) [13]. Both algorithms are an outcome of the Twitch's ACM MMSys 2020 Grand Challenge [8]. dash.js ships with a fully configurable sample player to evaluate the various options and settings related to low latency streaming [9].

4.1 Low on Latency+

Low on Latency+ is designed as a series of sophisticated yet robust player improvements for low latency live-streaming. LoL+ consists of five essential modules. The bitrate selection module implements a learning-based ABR algorithm to choose a suitable bitrate for each segment download. The ABR algorithm is based on an SOM model that considers multiple QoE metrics as well as bandwidth variability. The playback speed control module implements a hybrid algorithm that considers both the current latency and buffer level to control the playback speed. The throughput measurement module accurately calculates the throughout by removing the idle times between the chunks of a segment through a three-step algorithm. The QoE evaluation module computes the QoE considering five key metrics: selected bitrate, number of bitrate switches, rebuffering duration, latency and playback speed. Lastly, the weight selection module implements a two-step dynamic weight assignment for the SOM model features. [3] [15]

A fully configurable LoL+ sample player can be found in the sample section of dash.js [10].

4.2 Learn2Adapt

Learn2Adapt strikes to find a favorable balance between keeping the buffer as short as possible, while provisioning against its complete depletion. This is achieved by selecting the highest sustainable bitrate for each video fragment, that does not completely consume the buffer budget available at the time of request. L2A-LL is, in essence, an optimization solution with the objective of minimizing latency, while at the same time maximizing achievable video bitrate and ensuring uninterrupted and stable streaming. [13]

L2A-LL formulates the ABR optimization problem under an online (machine) learning framework, based on convex optimization. First, the streaming client is modelled by a learning agent, whose objective is to minimize the average buffer displacement of a streaming session. Second, certain requirements regarding the decision set (available bitrates) and constraint functions are fulfilled by a)

[8] https://github.com/twitchtv/acm-mmsys-2020-grand-challenge
[9] https://reference.dashif.org/dash.js/nightly/samples/low-latency/testplayer/testplayer.html
[10] http://reference.dashif.org/dash.js/nightly/samples/low-latency/lolp_index.html

allowing the learning agent to make decisions on the video bitrate of each fragment, according to a probability distribution and by b) deriving an appropriate constraint function associated with the upper bound of the buffer queue, that adheres to time averaging constraints. [15]

A fully configurable L2A sample player can be found in the sample section of dash.js [11].

5 COMMON MEDIA CLIENT DATA

CTA-5004 - Common Media Client Data (CMCD) defines data that is collected by the media player and is sent as a custom HTTP header or query parameter alongside each object request to a CDN. CMCD allows the correlation of CDN data with metrics collected on the client side. Session identifiers allow thousands of individual CDN logs to be interpreted as a single playback session. The metric values defined in the CMCD specification can be useful for log analysis, quality of service monitoring, and delivery enhancements. Using the client state information, CDNs can prioritize certain clients and cross-correlate performance problems with specific devices and platforms. In addition, CDNs can use information about upcoming segment requests signaled by the player to cache media segments at the edge ahead of the client request. [6] [2]

Since version 3.2.1 [12] dash.js is fully compliant with the CMCD specification. A separate CMCD model class is responsible for collecting the CMCD-specific information and parameters. For that purpose, hooks for internal events are registered and the internal state of the class is updated once CMCD relevant parameters are updated. Before requests to the CDN, a valid CMCD payload is generated and appended by either adding separate CMCD header fields or by appending the CMCD information as query parameters to the request url. dash.js ships with a sample illustrating the usage and configuration of CMCD [13].

6 DIGITAL RIGHTS MANAGEMENT

Digital rights management (DRM) is a way to protect copyrights for digital media. This approach includes the use of technologies that limit the copying and use of copyrighted works and proprietary software. [8]

A DRM system cooperates with the device's media platform to enable playback of encrypted content. Decrypted samples and the content keys are protected against potential attacks. [7] dash.js supports two DRM systems namely Widevine and Playready and also offers support for Clearkey protection. Clearkey is primarily used for client and media platform development and test purposes and does not offer the same level of security as DRM systems like Widevine and Playready.

6.1 MPD based license server signaling

DRM systems use the concept of license requests as the mechanism for obtaining content keys. [7] Since version 4.2.0 dash.js supports the signaling of license server URLs via MPD parameters. For that reason, the MPD author defines an *<dashif:Laurl>* element as part of the *ContentProtection>* descriptor. An example of such signaling is depicted in Listing 4

Listing 4: Signaling of a license server url via MPD paramaters [7]

```
<ContentProtection
        schemeIdUri="urn:uuid:d0ee2730-09b5-459f-8452-200e52b37567"
        value="FirstDRM 2.0">
        <dashif:Laurl>https://example.com/AcquireLicense</dashif:Laurl>
</ContentProtection>
```

6.2 Manifest-based key rotation

Key rotation is a prominent option to for periodic re-authorization of the client, as well as forcing rights to be reevaluated at program boundaries. For that reason, an update of the *default_KID* in the manifest file can be used to signal a change of the content key. [7] dash.js supports such manifest-based key rotation since version 4.2.0. Once the *default_KID* is changed, a new key session is created and a license requests is issued. When a valid license has been received, the newly created key session is updated to enable decryption of the content using the new content key.

7 CONCLUSION

The trend to consume high-quality videos over the internet lead to a high demand for sophisticated and robust video player implementations. In this paper, we introduced the latest additions and improvements to dash.js, an open source MPEG-DASH player provided by DASH-IF.

We discussed MPD patching as a mean to provide only non-redundant and mandatory MPD update information to the player, minimizing bandwidth and parsing requirements. Moreover, improvements to the multiperiod implementation were highlighted, focusing on prebuffering, the calculation of the DVR window and the transition between non-encrypted and encrypted periods. Gap and MSE error handling as essential mechanism to avoid playback stalling and to guarantee a robust player operation were introduced. Furthermore, improvements to the UTC synchronization were highlighted.

In the context of low-latency streaming, two new ABR algorithms were introduced, namely Low on Latency+ (LoL+) and Learn2Adapt (L2A). Both, LoL+ and L2A aim to overcome the challenge of finding the optimal video and audio quality when streaming with low latencies and small buffers.

Common Media Client Data (CMCD) as a mean to monitor and evaluate streaming sessions and to optimize CDN delivery was introduced. Finally, we discussed advances in the field of Digital Rights Management (DRM). For that reason, means to use the MPD in order to provide license server information and enable key rotation were discussed.

Future work in dash.js includes optimization of the existing throughput calculation and XML parsing functionalities. Moreover, we are looking into running MSE operations in webworkers, providing a new reference UI, adding support for Producer Reference Time (PRFT) boxes and enhancing the existing CMCD implementation with support for custom key/value pairs.

[11] http://reference.dashif.org/dash.js/nightly/samples/low-latency/l2all_index.html
[12] https://github.com/Dash-Industry-Forum/dash.js/releases/tag/v3.2.1
[13] https://reference.dashif.org/dash.js/nightly/samples/advanced/cmcd.html

ACKNOWLEDGMENTS

The authors would like to thank DASH-IF for their constant support and feedback on the dash.js project, especially Thomas Stockhammer, Will Law, Iraj Sodagar, Nicolas Weil, Romain Bouqueau and Ali Begen.

Moreover, the authors would like to thank the dash.js community for their contributions to the player code, especially Bertrand Berthelot, Nicolas Angot, Jesus Olivia, Dan Sparacio, David Evans, Robert Bryer, Jeremie Collet, Jean-François Cunat, Fritz Heiden, Vinay Rosenberg and Björn Altmann.

The authors would also like to congratulate the winners of the 2021 dash.js awards namely Bertrand Berthelot, Zachary Cava, Theo Karagkioules, Rufael Mekuria, Dirk Griffioen, Arjen Wagenaar, May Lim, Mehmet N. Akcay, Abdelhak Bentaleb, Ali C. Begen, Roger Zimmermann.

REFERENCES

[1] Chris Bass. 2021. *Streaming player shakedowns: Our setup to test internet streaming.* https://www.bbc.co.uk/rd/blog/2021-10-internet-streaming-test-network-conditions-dash

[2] Ali C. Begen, Abdelhak Bentaleb, Daniel Silhavy, Stefan Pham, Roger Zimmermann, and Will Law. 2021. Road to Salvation: Streaming Clients and Content Delivery Networks Working Together. *IEEE Communications Magazine* 59, 11 (2021), 123–128. https://doi.org/10.1109/MCOM.121.2100137

[3] Abdelhak Bentaleb, Mehmet N. Akcay, May Lim, Ali C. Begen, and Roger Zimmermann. 2021. Catching the Moment with LoL+ in Twitch-Like Low-Latency Live Streaming Platforms. *IEEE Transactions on Multimedia* (2021), 1–1. https://doi.org/10.1109/TMM.2021.3079288

[4] Bitmovin. 2021. Bitmovin Video Developer Report 2021. (2021). https://go.bitmovin.com/video-developer-report

[5] Zachary Cava. 2019. DASH Live Streaming at scale. https://dashif.org/docs/workshop-2019/16-Cava-2019-12-portland-video-meetup.pdf. Accessed: 2021–12-07.

[6] CTA-5004 2020. *Web Application Video Ecosystem–Common Media Client Data* . Specification. Consumer Technology Association. https://shop.cta.tech/products/web-application-video-ecosystem-common-media-client-data-cta-5004

[7] DASH-IF Interoperability Points;Part 6: Content Protection 2021. . Technical Report. https://dashif.org/docs/IOP-Guidelines/DASH-IF-IOPv5.0-Part6-REVIEW-20210805.pdf

[8] Juliana De Groot. 2018. *What is Digital Rights Management?* https://digitalguardian.com/blog/what-digital-rights-management

[9] Fraunhofer FOKUS. 2020. *dash.js - FAME maintains dash.js project source code.* https://www.fokus.fraunhofer.de/en/fame/news/dash.js

[10] DASH Industry Forum. 2020. *DASH Industry Forum - About.* https://dashif.org/about/

[11] Guidelines for Implementation:DASH-IF Interoperability Points 2018. . Technical Report. https://dashif.org/docs/DASH-IF-IOP-v4.3.pdf

[12] ISO/IEC FDIS 23009-1 4th edition 2019. *Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats* . Standard. International Organization for Standardization.

[13] Theo Karagkioules, Rufael Mekuria, Dirk Griffioen, and Arjen Wagenaar. 2020. Online Learning for Low-Latency Adaptive Streaming. In *Proceedings of the 11th ACM Multimedia Systems Conference* (Istanbul, Turkey) *(MMSys '20)*. Association for Computing Machinery, New York, NY, USA, 315–320. https://doi.org/10.1145/3339825.3397042

[14] Sandvine. 2019. The Global Internet Phenomena Report. (2019). https://www.sandvine.com/global-internet-phenomena-report-2019

[15] Daniel Silhavy. 2021. *Low Latency Streaming.* https://github.com/Dash-Industry-Forum/dash.js/wiki/Low-Latency-streaming

[16] Daniel Silhavy, Stefan Pham, and Stefan Arbanowski. 2017. Performance Considerations of HTML5-Based Dynamic Packaging for Media Streaming. In *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video* (Taipei, Taiwan) *(NOSSDAV'17)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3083165.3083172

[17] World Wide Web Consortium 2016. *Media Source Extensions.* World Wide Web Consortium. https://www.w3.org/TR/2016/REC-media-source-20161117/

[18] Bo Zhang, Thiago Teixeira, and Yuriy Reznik. 2021. *Performance of Low-Latency HTTP-Based Streaming Players.* Association for Computing Machinery, New York, NY, USA, 356–362. https://doi.org/10.1145/3458305.3478442