Low Latency Streaming and Multi DRM with dash.js

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

Stefan Pham
Fraunhofer FOKUS
stefan.pham@fokus.fraunhofer.de

Martin Lasak
Fraunhofer FOKUS
martin.lasak@fokus.fraunhofer.de

Anita Chen
Fraunhofer FOKUS
anita.chen@fokus.fraunhofer.de

Stefan Arbanowski
Fraunhofer FOKUS
stefan.arbanowski@fokus.fraunhofer.de

ABSTRACT

Video streaming applications account for 60% of today's global internet traffic. The trend to consume videos over the internet lead to a high demand for sophisticated and robust video players. dash.js is an open source DASH player of the DASH-Industry-Forum written in JavaScript utilizing the native browser APIs Media Source Extensions (MSE) and Encrypted Media Extensions (EME). This paper gives a general overview of the player and presents two specific features namely low-latency streaming and multi DRM playback. For that purpose, we illustrate how CMAF chunks in combination with the corresponding dash.js APIs and additional manifest parameters enable low latency streaming in the browser. For DRM support we focus on the interaction between dash.js, the EME and the underlying Content Decryption Module (CDM) of the browser.

CCS CONCEPTS

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

KEYWORDS

MPEG-DASH, dash.js, Media Source Extensions, Encrypted Media Extensions, CMAF

ACM Reference Format:
Daniel Silhavy, Stefan Pham, Martin Lasak, Anita Chen, and Stefan Arbanowski. 2020. Low Latency Streaming and Multi DRM with dash.js. In *11th ACM Multimedia Systems Conference (MMSys'20), June 8–11, 2020, Istanbul, Turkey*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3339825.3394936

1 INTRODUCTION

In 2019, 60% of the global application internet traffic was caused by video streaming applications [5]. Streaming providers such as Netflix and YouTube accounted for 12.6% and 8.7% of the global application traffic, respectively [5]. 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 2019, HLS is used by 79% of the streaming providers while MPEG-DASH reaches 58% adaption [1].

The trend in consuming videos over the internet leads to a high demand for sophisticated and robust video players. Poor player implementation will result in bad user experience and potential loss of customers. One of the major challenges in terms of player development is the heterogeneity of target platforms and devices, ranging from gaming consoles to Smart TVs, mobile devices and desktop browsers. In order to support a wide variety of these platforms and devices, 55% of streaming providers use an open source codebase for their players [1].

In this paper, we introduce dash.js, a free, open source MPEG-DASH player that serves as a JavaScript reference client for implementing production grade DASH players. It is available on GitHub [1] and NPM [2] under the BSD license. A hosted version of the reference sample page is also available [3]. dash.js is an outcome of Dash Industry Forum (DASH-IF), a group of leading streaming companies that promote and catalyze the adoption of MPEG-DASH and help transition it from a specification into a real deployment [2]. The dash.js player is written in JavaScript and relies on the Media Source Extensions (MSE) and the Encrypted Media Extensions (EME) defined by World Wide Web Consortium (W3C). Thus, dash.js can be used on all platforms, which offer support for MSE and EME. A majority of the common target platforms support both MSE and EME, which makes dash.js a prominent option for a production grade player. Having a single player which runs on all of the required target devices and platforms eliminates the need in maintaining different codebases and as a result, potentially saves time and resources. Moreover, clients using dash.js benefit from the active development of the player and the contributions from the community.

The dash.js project has been underway since November 2012. By February 2020, 125 developers have contributed to the players' source code. The project has 1,200 forks, 3,100 stars and is used and watched by 329 and 270 GitHub members, respectively. In 2019, the project had 315,309 NPM downloads. By the time this paper was written, the latest version of the dash.js player is 3.1.0.

dash.js offers a wide set of features related to adaptive media streaming. This includes:

- Support for Video on Demand (VOD) and live playback of MPEG-DASH and Microsoft Smooth Streaming (MSS) assets.

MMSys'20, June 8-11, 2020, Istanbul, Turkey

[1] https://github.com/Dash-Industry-Forum/dash.js
[2] https://www.npmjs.com/dashjs
[3] http://reference.dashif.org/dash.js/v3.1.0/samples/dash-if-reference-player/index.html

- Low latency MPEG-DASH playback based on Common Media Application Format (CMAF) chunks.
- Support for multiple Digital Rights Management (DRM) systems as well as different EME versions.
- Adaptive bitrate switching based on throughput, dropped frames and buffer level. The different adaptive bitrate (ABR) rules can be dynamically turned off and on. Moreover, the addition of customized ABR rules is possible.
- Multi-period support: includes support for codec switch between different periods.
- Support for multiple subtitle and captions formats namely Timed Text Markup Language (TTML), embedded CEA-608 closed captions (CEA-608), Web Video Text Tracks Format (WebVTT) and Internet Media Subtitles and Captions (IMSC-1) in both text and picture mode.
- Support for inband(EMSG box) and inline events (events included in the Media Presentation Description (MPD)).
- Gap jumping
- Cross-browser compliance: tested on various browsers to provide a consistent cross-browser experience.

This paper discusses two crucial features, namely, "low latency streaming with CMAF" and "multi DRM support". The remainder of the paper is structured as follows:

In Section 2, low latency streaming with CMAF is introduced. The general working principle of CMAF chunks for low latency streaming is illustrated and the benefits compared to conventional ISO base media file format (ISOBMFF) segments are highlighted. In addition, this section describes the required signaling of low latency mode in the manifest, and the mechanisms to set the target latency and configure the catchup mechanism with dash.js.

Section 3 explains the need for DRM systems and the role that EME plays as an interface between the Content Decryption Module (CDM) and the browser. Subsequently, the concrete interaction between the dash.js player and the EME is illustrated. In addition, the specific requirements to enable hardware DRM are presented.

Both Sections 2 and 3 have a similar structure. First, the basics behind the corresponding technologies are explained. Next, the concrete implementation in dash.js is detailed, as well as highlighting important implementation aspects.

The paper closes with a short summary and an outlook of upcoming dash.js features.

2 LOW LATENCY STREAMING WITH CMAF IN DASH.JS

This section provides an overview of the CMAF media file format, its mechanism for low latency live streaming, and how this works in combination with dash.js.

2.1 Common Media Application Format

CMAF is essentially another media container based on ISOBMFF and fragmented MP4. The major advantage of CMAF, in comparison to classic media containers like ISOBMFF and MPEG transport stream (MPEG-TS), is that it can be referenced from HLS and MPEG-DASH manifest files. Consequently, media streaming content would only need to be encoded and packaged once (within the CMAF container) and can then be streamed to all major platforms. Content providers are no longer forced to create and store separate media files with an MPEG-TS container for HLS and an ISOBMFF container for MPEG-DASH. As a result, CMAF immediately cuts the storage and packaging costs in half while doubling CDN efficiency at the same time.

In addition, CMAF provides the necessary tools for low latency live streaming. CMAF introduces the concept of "chunks". On a high level, a classic ISOBMFF segment consists of one "moof" box and one "mdat" box. With CMAF chunks, the segment now has multiple such boxes that allow the client to access the media data before the segment finishes. The benefits of the chunked mode become more obvious when looking at the example depicted in Figure 1.

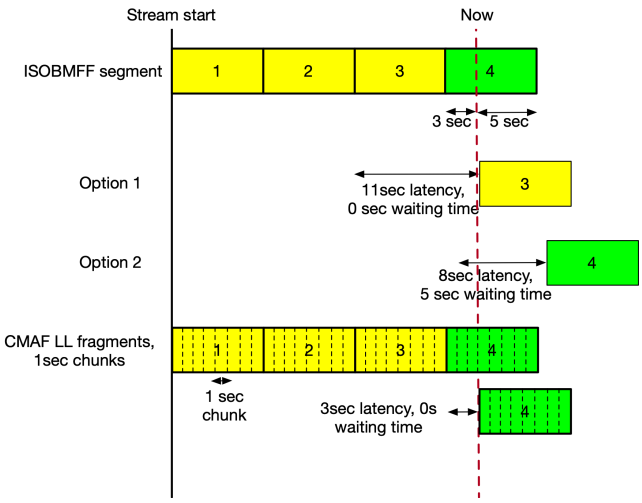

Figure 1: Latencies achieved with classic ISOBMFF segments and CMAF chunks.

In this example, each segment has a duration of 8 seconds. The current wall clock time (now time) maps to the third second of segment number four. A player entering the presentation at the current wall clock time playing classic media segments, has two options:

- Option 1: since segment four is not completed, the player can start with segment three. That way, it ends up being 11 seconds behind the live edge – 8 seconds coming from segment three, and 3 seconds coming from segment four.
- Option 2: the player waits 5 seconds for segment four to finish and immediately starts downloading and playing it. That way, the players ends up with 8 seconds of latency and a waiting time of 5 seconds.

With CMAF chunks, on the other hand, the player is able to play segment four before it is completely available. In the example above, the CMAF chunks have a 1 second duration, which leads to eight chunks per segment. Typically, only the first chunk contains an Instant Decoder Refresh (IDR) frame and therefore, the player would always need to start the playback from the beginning of a segment. Consequently, being 3 seconds into segment four results

in a latency of 3 seconds, which is less than what could be achieved with classic ISOBMFF segments. It is also possible to fast decode the first chunks by increasing the playback rate and play even closer to the live edge.

2.2 CMAF low latency with dash.js

Since version 2.6.8, dash.js supports low latency streaming using CMAF. The DASH-IF offers two sample streams with low latency support generated by the DASH-IF live simulator [4] [5]. A concrete example on how to use dash.js in low latency mode is provided in this section.

2.2.1 Signaling low latency mode in the manifest file. A way to signal to the client that the segments are chunked and available prior to completion is required. For that reason, two new attributes are introduced in the MPD [3]:

- @availabilityTimeComplete (ATC): specifies if segments of all associated representations are completed at the adjusted availability start time. If the value is set to "false", then the client can infer that the segment is available at its announced location prior to completion.
- @availabilityTimeOffset (ATO): provides the time in how much earlier the segments are available compared to their computed availability start time (AST).

By setting ATC to "false", the packager signals to the client that the segments are available prior to completion. Using the ATO attribute, the packager can specify how much earlier (in comparison to normal completion time) the segments can be accessed. In the example depicted in 1, each segment has a duration of 8 seconds and the ATO is set to 7 seconds. This means that the segments have a chunk duration of 1 second and the first segment is available 7 seconds before its usual completion time.

2.2.2 Setting the target latency. The first option to specify when streaming with dash.js in low latency mode is the target latency. Target latency defines how close dash.js plays to the live edge. In an ideal scenario, the target latency would be 0, and dash.js would play directly at the live edge. Unfortunately, reducing the target latency always comes with the trade-off in reducing the size of the media buffer. A small media buffer, on the other hand, is vulnerable to bandwidth fluctuations. In the worst case scenario, the player runs into an empty buffer and playback is stalled. Thus, switching to a small latency has a significant influence on the stability of the stream and should always be evaluated carefully before production deployment.

Setting the target latency with dash.js is very straightforward and requires one line of code, as illustrated in Listing 1.

```
player.updateSettings({
            'streaming': {
                'liveDelay': 3
            }
        });
```

Listing 1: Setting the target latency in dash.js

A live delay of 2-3 seconds is a good trade-off between playing closely to the live edge and maintaining a stable buffer.

[4] https://tinyurl.com/tn55bqb
[5] https://tinyurl.com/thfnj9u

2.2.3 Configuring the catchup mechanism. In the example illustrated in Figure 1, a latency of 3 seconds for the CMAF fragment with a chunk duration of 1 second was achieved. However, there is an easier way to get even closer to the live edge: by simply increasing the playback rate. Increasing the playback rate is also useful in scenarios in which a latency deviation between the target and real latency occurs. In dash.js, this process is called "catch-up mechanism", which is controlled by two parameters:

By setting the liveCatchUpMinDrift attribute (in seconds), the minimum latency deviation allowed before activating the catch-up mechanism is defined. Again, the concrete call is a one-liner, as shown in Listing 2.

```
player.updateSettings({
            'streaming': {
                'liveCatchUpMinDrift': 0.1
            }
        });
```

Listing 2: Setting minimum latency deviation allowed before activating the catch-up mechanism in dash.js

In this example, the allowed drift is set to 0.1 seconds.

In addition to the allowed drift, the desired catchup playback rate needs to be defined. Again, one line of code is required:

```
player.updateSettings({
            'streaming': {
                'liveCatchUpPlaybackRate': 0.5
            }
        });
```

Listing 3: Setting the catchup playback rate for low latency streaming in dash.js

In this example, the playback rate is increased by 50% to 1.5 if the catch-up mechanism is activated.

Increasing or decreasing the playback rate without notification can lead to a poorer viewing experience. For some viewers, it may look like the stream is interrupted, especially because the audio is affected as well. A user-friendly solution for this is to mute the audio or inform the user with a notification popup.

2.2.4 Additional configuration. In order to use the low latency feature, the browser running dash.js needs to support the Fetch API and HTTP 1.1 chunked transfer encoding. The combination of both allows access to the media data prior to the availability of the media segment.

3 DIGITAL RIGHTS MANAGEMENT IN DASH.JS

Streaming providers are required to protect media assets against piracy. The common approach in preventing unauthorized redistribution of media files is by using DRM systems. Depending on the target platform, different DRM systems are required. The three main DRM systems in regards to adaptive video streaming are Google Widevine, Microsoft Playready and Apple Fairplay.

EME is the API that enables playback of DRM protected content in the browser. It provides the necessary function calls to discover and interact with the underlying DRM system.

Like any other API, EME evolved over a period of time and the current version is a vastly different compared to the one in 2013. While desktop and mobile browsers are frequently updated, some

embedded devices and set-top boxes are still running an outdated or customized version of the EME. For that reason, a sophisticated player would detect the EME version on the client and trigger the correct API functions.

3.1 License acquisition with multiple EME versions

By default, dash.js comes with support for three different versions of EME:

- ProtectionModel_01b.js: initial implementation of the EME by Google Chrome (prior to version 36). This EME version is not promised-based and uses outdated or prefixed events like "needkey" or "webkitneedkey".
- ProtectionModel_3Feb2014.js: implementation of EME APIs as of 3 Feb 2014. Implemented by Internet Explorer 11 (Windows 8.1).
- ProtectionModel_21Jan2015.js: most recent EME implementation. Latest changes in the EME specifications are added to this model. It supports promised-based EME function calls.

3.2 How to select the correct EME version

dash.js injects the correct EME version once the player is initialized, as illustrated in Listing 4

```
if ((!videoElement || videoElement.onencrypted !==
    undefined) &&
        (!videoElement || videoElement.mediaKeys !==
            undefined)) {
        return ProtectionModel_21Jan2015
    }

else if (getAPI(videoElement,
    APIS_ProtectionModel_3Feb2014)) {
        return ProtectionModel_3Feb2014
    }

else if (getAPI(videoElement, APIS_ProtectionModel_01b))
    {
        return ProtectionModel_01b
    }
```

Listing 4: Initialization of the correct EME version in dash.js

For means of simplicity, the actual instantiation of protection models was removed, so the player checks for the correct EME version in a reversed order. That way, the latest available EME version is selected and the appropriate ProtectionModel is returned to the controlling entity (ProtectionController.js). At this point, it is also possible to add customized protection models in order to support customized versions of the EME.

3.3 The basic EME flow

This section describes the entire license acquisition process, as performed by dash.js using the latest version of the EME (corresponding to ProtectionModel_21Jan2015).

3.3.1 Detecting encrypted content. In general, the information as to if and how the content is encrypted can either be a part of the manifest file and/or be embedded in the media segments. In this example, we assume that the DRM information is embedded in the media segments. If the content is encrypted, the dash.js player receives an encrypted event from the browser. By registering for that type of event, the player can pass the DRM initialization data to a callback function:

```
case 'encrypted':
  if (event.initData) {
        let initData = event.initData
        eventBus.trigger(events.NEED_KEY,
                    {key:new NeedKey(initData, event.
                        initDataType)});
    }
```

Listing 5: The encrypted event of the browser provides the necessary DRM initialization data.

By parsing the initialization data, the player can identify which DRM systems can be used in order to decrypt the content. For instance, one content may only support a Playready DRM, while another one supports both Playready and Widevine.

3.3.2 Selecting the right DRM system. Depending on the underlying platform and browser, multiple DRMs may be available – for example, both Playready and Widevine at the same time. Since only one DRM system is required to decrypt the content, dash.js allows a prioritization of DRM systems. The respective call that needs to be completed before delivering the manifest to the player is depicted in Listing 6.

```
const protData =
    "com.widevine.alpha": {
        "serverURL": "https://drm-widevine-licensing.
            axtest.net/AcquireLicense",
        "priority": 1
    },
    "com.microsoft.playready": {
        "serverURL": "https://drm-playready-licensing.
            axtest.net/AcquireLicense",
        "priority": 0
    };
player.setProtectionData(protData);
```

Listing 6: dash.js allows the prioritization of available DRM systems.

In this scenario, two valid configurations are defined, one for Playready DRM, and one for Widevine DRM. Due to the prioritization order, the availability of a Widevine DRM is checked before the availability of a Playready DRM.

3.3.3 Requesting access to the DRM system. Before decrypting the content, the player needs to check if the platform supports one of the specified DRM systems or not. For that purpose, the requestMediaKeySystemAccess() function of the EME is used. A successful call to this function will return a MediaKeySystemAccess object:

```
navigator.requestMediaKeySystemAccess(systemString,
    configs)
  .then(function (mediaKeySystemAccess) {
          keySystemAccess.mksa = mediaKeySystemAccess;
          })
    .catch(function(error) {
        // configuration is not supported
    })
```

Listing 7: Requesting access to the DRM system with requestMediaKeysSystemAccess.

3.3.4 Generating a payload for the license request. After dash.js has selected the correct DRM system, the previously received MediaKeySystemAccess object is used to create MediaKeys and assign them to the HTML5 video element. Later on, the MediaKeys will be used to decrypt the content:

```
keySystemAccess.mksa.createMediaKeys()
    .then(function (mkeys) {
            mediaKeys = mkeys;
            videoElement.setMediaKeys(mediaKeys)
                  .then(function () {
                   });
            }
```

Listing 8: Creating the Mediakeys which will be used for decryption of the content

In order to receive a valid license for the content, a CDM specific payload needs to be added to the license request. For that purpose, a MediaKeySession is created, in which dash.js calls the generateRequest() function.

```
const session = mediaKeys.createSession(sessionType);
session.generateRequest(dataType, initData)
    .then(function () {
         // Request generated
        })
    .catch(function (error) {
        // Error
        });
```

Listing 9: Generating a request to receive the CDM specific payload for the license request.

The browser will forward the request to the underlying CDM. As a result, the CDM generates the payload for the license request.

3.3.5 Sending the license request. When the CDM has generated the required payload, the data is forwarded to the browser. By registering for the message event, the player is able to grab the needed data:

```
case 'message':
   let message = ArrayBuffer.isView(event.message) ?
        event.message.buffer : event.message;
```

Listing 10: Receiving the payload for the license request from the CDM

Finally, the license request can be issued using the reqPayload derived from the previous key message:

```
doLicenseRequest(url, reqHeaders, reqMethod, responseType
    , withCredentials, reqPayload,
    LICENSE_SERVER_REQUEST_RETRIES, timeout, onLoad,
    onAbort, onError);
```

Listing 11: Issuing a license request

3.3.6 Working with the license response. If the license server returns a valid license, the final step is to update the MediaKeySession with the data received from the license server.

```
session.update(message).catch(function (error) {
});
```

Listing 12: Updating the MediaKeySession

At this point, the player has everything it needs to play the content. The rest is up to the browser and CDM.

Table 1: Mapping of EME levels to Widevine security level [4]

EME Level	Robustness Level	Widevine Security Level
1	SW_SECURE_CRYPTO	3
2	SW_SECURE_DECODE	3
3	HW_SECURE_CRYPTO	2
4	HW_SECURE_DECODE	1
5	HW_SECURE_ALL	1

3.4 Enabling Hardware DRM on Android Chrome using Encrypted Media Extensions

DRM systems offer different levels of security. Taking Google's Widevine DRM system as a reference, three different security levels are defined:

- Security Level 1 (L1): complete processing is performed in a Trusted Execution Environment (TEE). This level refers to a hardware DRM.
- Security Level 2 (L2): cryptography is performed within the TEE. The video processing is done through separate video hardware or software. This level also refers to hardware DRM.
- Security Level 3 (L3): no TEE is present in the device. Decryption is typically performed directly in the browser. This level refers to software DRM.

In several cases, only devices with a security level of L1 are allowed to play Ultra HD content. Thus, an interesting question arises in regards to how to check if a device supports an L1 Widevine DRM or not.

3.4.1 Hardware DRM and EME - the theory. As described in Section 3.3.3, EME's requestMediaKeySystemAccess function is used to detect which DRM systems are available, along with its supported configurations. The EME defines five different levels which can be directly mapped to its respective Widevine security level, as illustrated in Table 1.

For example, EME level 1 maps to a Widevine security level of 3.

The EME level is not specified directly in the requestMediaKeySystemAccess call. Instead, the robustness level parameter is used (see Table 1). The complete invocation of the initial EME call includes a configuration array that combines all required parameters.

Typically, not only is the video track encrypted, but the audio track as well. Widevine recommends using different encryption keys for both tracks. Video tracks are much more valuable and as a result, platforms only support Widevine L3 audio tracks. Following that, a sample configuration for the requestMediaKeySystemAccess call can include the following values:

```
const config = [
  {
    "initDataTypes": [
      "cenc"
    ],
    "persistentState": "optional",
    "distinctiveIdentifier": "optional",
    "sessionTypes": [
      "temporary"
```

```
    ],
    "audioCapabilities": [
      {
        "robustness": "SW_SECURE_CRYPTO",
        "contentType": "audio/mp4;codecs="mp4a.40.2""
      }
    ],
    "videoCapabilities": [
      {
        "robustness": "HW_SECURE_ALL",
        "contentType": "video/mp4;codecs="avc1.42800C""
      }
    ]
  }
]
```

Listing 13: Sample configuration that is handed to the requestMediaKeySystemAccess function

In this example, the video track requires at least a Widevine level 1 DRM, while audio only needs level 3. One thing to keep in mind is that changing parameters like persistentState, distinctiveIdentifier or the contentType have a significant influence on the result of the call. The underlying CDM may reject the configuration if one of these settings is not supported, regardless of whether a hardware DRM is supported or not.

3.4.2 Hardware DRM and EME - Practical Tests. Based on the previously described code, we conducted a small test, in which we used a Samsung Galaxy S9 with Android 9 and Chrome 75.0.3770.101, and a HTC OnePlus 5T with Android 8.1 and Chrome 75.0.3770.101.

With the configuration described in Listing 13, the promise returned by the requestMediaKeysSystemAccessCall is successfully resolved on both devices. Unfortunately, this does not necessarily mean that the device really supports hardware DRM. On one of our test devices, we encountered an error when trying to create the MediaKeys afterwards. Therefore, when checking if a device supports hardware DRM or not, the MediaKeys need be checked as well. This results in the code shown in Listing 14.

```
navigator
        .requestMediaKeySystemAccess(keySystem, config)
        .then((keySystemAccess) => {
          return keySystemAccess.createMediaKeys();
        })
        .then(() => {
          // It works
        })
        .catch((e) => {
          // no UHD on this device
        });
```

Listing 14: The require EME calls to check for a Hardware DRM.

4 CONCLUSION

This paper introduced dash.js, the open source MPEG-DASH player of DASH-IF that is written in JavaScript and utilizes the native browser APIs MSE and EME. dash.js offers a wide variety of features, like VOD and live playback, MSS to MPEG-DASH conversion, as well as support for multi-period and buffer- and throughput-based adaptive bitrate algorithms. Within the scope of this paper, two specific use cases, namely low-latency streaming and multi-DRM playback, were presented.

In the context of low latency streaming, CMAF chunks, used as a means to access the media data of a segment, and prior to the segment being completely available on the Content Delivery Network (CDN), were introduced. Furthermore, the two MPD attributes, @availabilityTimeComplete and @availabilityTimeOffset, to specify if and how much earlier a segment is available at its announced location, were presented as well. Subsequently, the concrete low latency implementation in dash.js was explained, focusing on the three relevant API functions of the player that control the parameters for target latency, latency deviation and catch-up rate. Finally, potential drawbacks, like irritating sound effects (due to an increased playback rate in catch-up mode), and additional requirements, like support for the Fetch API and HTTP 1.1 chunked transfer encoding, were outlined.

For DRM support in dash.js, the general requirement for streaming providers in protecting their content against piracy was highlighted. In browser-based environments, EME is the interface for accessing DRM's underlying functionality. dash.js offers support for different EME versions in order to enable playback of DRM protected content on legacy devices and latest browser versions. The complete license acquisition process in dash.js comprises of multiple steps. This process includes passing a valid configuration to the CDM, creating a key session and its corresponding media keys, and generating valid request data to deliver to the license server. These steps were detailed in Section 3. In addition, the implications of different DRM security levels on the corresponding EME calls were investigated. A practical test case was also provided in order to show that checking the availability of a hardware DRM requires at least two different EME calls.

Future work in the context of dash.js development includes: support for offline playback, implementation of the Common Media Client Data specification (CTA 5005). Furthermore, multi-period and live implementations for SegmentTimeline, as well as gap management, will be improved. Additionally, the ABR algorithm for low latency streaming is to be revised.

REFERENCES

[1] Bitmovin. 2019. Bitmovin Video Developer Report 2019. (2019). https://bitmovin.com/bitmovin-2019-video-developer-report-av1-codec-ai-machine-learning-low-latency

[2] DASH Industry Forum. [n.d.]. *DASH Industry Forum - About.* https://dashif.org/about/

[3] 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.

[4] Microsoft. 2019. *Offline Widevine streaming for Android.* https://docs.microsoft.com/en-us/azure/media-services/previous/offline-widevine-for-android

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