Public Review for

Common Media Client Data (CMCD): Initial Findings

A. Bentaleb, M. Lim, M. N. Akcay, A. C. Begen, R. Zimmermann

In September 2020, the Consumer Technology Association (CTA) published the CTA-5004: Common Media Client Data (CMCD) standard. With CMCD, a media client can send information to the content delivery network servers along with the object requests. This information can be quite useful in log association/analysis, quality of service/experience monitoring and delivery enhancements. This paper demonstrates how CMCD can be used to improve the performance of adaptive streaming. The work leverages the recent dash.js implementation of CMCD, coupled with an NGINX module to parse CMCD information and a bandwidth allocation logic (at the server) using the CMCD data. Specifically, the authors developed a proof-of-concept system that conforms to the client and server-side CMCD specification. On the server side, the authors implemented a CMCD-aware HTTP server using NGINX that runs a buffer-aware bandwidth allocation algorithm, and on the client side, they used the dash.js client with CMCD support on headless Chrome browser instances, instrumented using Puppeteer. The authors evaluated this system in a number of multi-client shared-network scenarios through network emulation. The results show a great potential for CMCD. The authors emphasize that the more streaming client implementations and content delivery networks start adopting this standard, the greater the benefits will be. Given the recency of the CMCD standard, this study must be one of the first on this topic, and we are delighted to have it as part of the technical program at NOSSDAV'21. Moreover, we applaud the effort of the authors for making all their code freely and openly available in a clean and easy-to-use format. We believe that this will be a useful asset to other researchers working in this area, facilitating further research building on this work.

Public review written by
Andra Lutu
Telefonica Research, Spain

Common Media Client Data (CMCD): Initial Findings

Abdelhak Bentaleb[★], May Lim[★], Mehmet N. Akcay[+], Ali C. Begen[+] and Roger Zimmermann[★]
[★]National University of Singapore, [+]Ozyegin University
{bentaleb,maylim,rogerz}@comp.nus.edu.sg, necmettin.akcay@ozu.edu.tr, ali.begen@ozyegin.edu.tr

ABSTRACT

In September 2020, the Consumer Technology Association (CTA) published the CTA-5004: Common Media Client Data (CMCD) specification. Using this specification, a media client can convey certain information to the content delivery network servers with object requests. This information is useful in log association/analysis, quality of service/experience monitoring and delivery enhancements. This paper is the first step toward investigating the feasibility of CMCD in addressing one of the most common problems in the streaming domain: efficient use of shared bandwidth by multiple clients. To that effect, we implemented CMCD functions on an HTTP server and built a proof-of-concept system with CMCD-aware dash.js clients. We show that even a basic bandwidth allocation scheme enabled by CMCD reduces rebuffering rate and duration without noticeably sacrificing the video quality.

CCS CONCEPTS

• **Multimedia information systems** → *Multimedia streaming*.

KEYWORDS

CMCD, adaptive streaming, DASH, HLS, CDN, bandwidth allocation, ABR.

ACM Reference Format:
Abdelhak Bentaleb, May Lim, Mehmet N. Akcay, Ali C. Begen and Roger Zimmermann. 2021. Common Media Client Data (CMCD): Initial Findings. In *Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'21), Sept. 28-Oct. 1, 2021, Istanbul, Turkey*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458306.3461444

1 INTRODUCTION

With rapid demand growth for premium streaming services, viewer expectations for high media quality and low rebuffering are constantly increasing. Regardless of the device and type of the network used, an HTTP adaptive streaming (HAS) client runs an adaptive bitrate (ABR) scheme to select a suitable representation (*e.g.*, resolution and bitrate) that fits the transient network conditions during the streaming session. The ABR trades off the streaming quality with rebuffering rate/duration. Today, all streaming services (free or premium, small or large scale) use content delivery networks (CDN) for better reach, lower latency and higher quality, that is, to manage that trade-off better.

The streaming performance of, and hence, the viewer quality-of-experience (QoE) delivered by different CDN servers can vary substantially. A server that is performing well for one client watching a content does not necessarily deliver a good performance for other clients watching the same or a different content. Also, a sudden increase in the number of clients streaming a live event, a scenario referred to as flash crowd, may have a significant transient impact on the CDN performance. In addition, the network conditions might change unexpectedly. Given that a streaming client keeps a limited amount of media in its playback buffer, the client needs to react in real time and make the right decision(s) (*e.g.*, switching to a more appropriate representation, redirecting to a different server in the CDN or switching to another CDN) at the right time. This is essential to maintain a good viewer experience.

The everlasting task has been to meet the evolving quality expectations of diverse viewers with devices of heterogeneous capabilities. Approaches considered to date range from client and server-side solutions, to network-based or hybrid ones [13]. For instance, researchers and developers focused on developing better rate adaptation and transport-layer algorithms [33, 36], improving CDN and server selection rules [5, 17], and the use of multiple CDNs [15, 19], developing in-network and data-driven networking approaches [12, 18, 28] as well as using centralized entities to improve media delivery based on information coming from various components [24, 27, 32]. The use of a control plane framework and enabling a communication platform between different streaming elements was standardized by MPEG, a method which is called Server and Network Assisted DASH (SAND) [3], published as ISO/IEC 23009-5 in 2017 after two years of development.

Generally speaking, information exchange is useful. It is most useful, however, when the information is relevant, actionable and up-to-date. Thus, in a system using the SAND concepts, an important question is what information is relevant and actionable. While the SAND standard offered a framework to enable communications and information exchange, it purposely left this question for further research and innovation.

In September 2020, the Consumer Technology Association (CTA) finalized a specification that attempted to answer this question. The specification is called CTA-5004: Common Media Client Data (CMCD) [1]. CMCD defines data that is collected by a media client and sent to the CDN along with its HTTP requests. At minimum, as depicted in Figure 1, session identification enables CDN logs to be combined per media session and correlated with client logs leading to a clearer picture of CDN's delivery performance, player software issues and viewer experience. Additionally, CDN servers can fine-tune the midgress traffic by intelligently reacting to the time constraints implicit in media segment requests. Using prefetch hints, CDNs can also make segments available at the edge ahead of a request, which improves delivery performance. Besides, buffer level information is instrumental in prioritizing the requests.

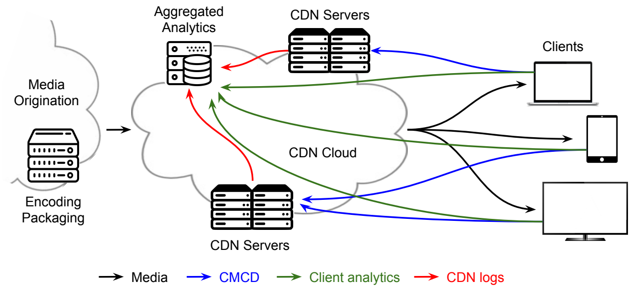

Figure 1: The CMCD-enabled HAS-based media delivery.

We make two contributions in this paper:

(1) This is the first study investigating the feasibility of the newly rectified CMCD specification and demonstrating its capabilities in improving viewer experience in one of the well-known problematic scenarios [7]. We evaluated the CMCD-aware system in a multi-client shared-network scenario through network emulation. We varied the number of clients to emulate an access link with 5–10 and an aggregation link with 20–30 concurrent streaming sessions.
(2) We implemented a proof-of-concept system that conforms with the client and server-side CMCD specification [1]. Our system consists of the open-source dash.js clients [20] and NGINX [4] server with a JavaScript module (NJS application). The entire system is publicly available at [37] to allow the scientific community and industry to debate its merits and perform further investigations.

In the rest of this paper, we highlight the related work in Section 2, provide a technical overview of CMCD in Section 3, present our CMCD-aware system in Section 4 and its evaluation in Section 5, and conclude the paper with future directions in Section 6.

2 RELATED WORK

2.1 Client-Side Solutions

To cope with dynamic network conditions, most streaming applications run a custom client-side ABR scheme that selects an appropriate quality level at decision epochs. The last decade has seen a large number of ABR scheme proposals at various levels of sophistication. Here, we mention a few of them and the interested readers are referred to [13]. Different techniques use throughput measurements [14, 31], playback buffer occupancy [26, 40], control theory [39] or game theory [11]. Other methods combine different heuristics (*e.g.*, throughput and buffer) [42], are congestion control aware [36] or use learning-based techniques [33].

2.2 Server-Side Solutions

Server-side solutions use a rate control technique at the server without any cooperation from the client. Therefore, the client's ABR scheme is implicitly controlled by the server's rate control. To that end, some methods [8, 25, 30] used traffic shaping, a tracker-assisted adaptation strategy [22], feedback control theory [21] or a multi-source solution [19]. This study can also be considered as a server-side solution, although the server in this case acts based on data provided by the client.

2.3 Network-Based Solutions

Network-based approaches have long been a focus of intense research. They are based on data collections from various network entities to aid in media delivery optimization. Network-based solutions can be further classified into: (*i*) In-network solutions: Some papers use software-defined networking (SDN) for bitrate guidance to assist clients in their ABR decisions [12], rate allocation [35], finding the best delivery path to re-route video traffic [16], or enabling multi-path capabilities [9]; (*ii*) SAND solutions: SAND [3] implements a control plane that defines asynchronous communication interfaces including between client-to-network, network-to-client and network-to-network. It enables the collection of various status information from network entities involved in media delivery including servers, caches, clients and other network entities along the media path. All this data is stored on a centralized server that helps improve media delivery and assist clients in their ABR decisions (see *e.g.*, [34, 38]); (*iii*) Data-driven solutions: A data-driven technique mixes SAND with AI capabilities for improved decision making. It uses a logically centralized controller that maintains a global view of real-time network conditions by gathering QoE metrics from many media sessions and then uses this global view to make suitable decisions regarding the ABR of individual sessions. Among existing works are [24, 28]; and (*iv*) Commercial solutions: *E.g.*, Conviva, Datazoom, Nice People at Work and SSIMWAVE.

3 TECHNICAL OVERVIEW OF CMCD

With CMCD a media client can convey information to a CDN server using one of three methods [1]: through a custom HTTP request header, as a query argument or within a JSON object. While different systems may prefer or require a specific method, here we use the query argument method where the CMCD parameters are listed in ⟨key, value⟩ pairs and carried as a query argument in the request:

```
Object-URL?CMCD=<cmcd-key1=value1,...,keyN=valueN>
```

CMCD defines two types of identifiers: `Content ID` and `Session ID`. Both are unique strings, with the former used to identify the content that is being streamed while the latter is a session identifier that is randomized for privacy reasons. They allow the CDN to correlate the CMCD data sent at different times and by different clients, and identify the source(s) of a problem. The specification describes several parameters, some of which are listed in Table 1.

Upon receiving a valid CMCD parameter, the server interprets its value according to the description in [1]. The server organizes the received CMCD parameters for different sessions based on the session IDs and then performs a set of suitable actions. Unknown parameters or malformed ones are categorically ignored.

4 SYSTEM IMPLEMENTATION

To investigate the feasibility of CMCD and test its capabilities in the context of video delivery, we implemented a proof-of-concept system. The system, depicted in Figure 2, consists of an NGINX server acting as the HTTP server with CMCD capabilities and a varying number of CMCD-aware clients based on the dash.js client [20]. The source code for the entire system is available at [37].

Table 1:Example parameters from [1] and the ones we added.

Parameter	Key	Type	Unit
`Encoded bitrate`	br	integer	Kbps
`Buffer length`	bl	integer	ms
`Buffer starvation`	bs	boolean	-
`Deadline`	dl	integer	ms
`Measured throughput`	mtp	integer	Kbps
`Requested max. throughput`	rtp	integer	Kbps
`Object type`	ot	token	-
`Max buffer` (new)	com.example-bmx	integer	ms
`Min buffer` (new)	com.example-bmn	integer	ms

4.1 CMCD-Aware Clients

We used the initial CMCD implementation offered by the dash.js client (v3.1.3) to build our CMCD-aware client. This implementation is mainly given in the CmcdModel.js class, which is responsible for collecting the required values for the set of the CMCD parameters that are retrieved from different classes of the client. For example, CmcdModel.js acquires the current `buffer length` (bl) from BufferController.js and the `measured throughput` (mtp) from ThroughputHistory.js. After collecting the required values, it generates the final CMCD query string to be sent with the HTTP GET request using HTTPLoader.js. Figure 3 shows the workflow of the CmcdModel.js class.

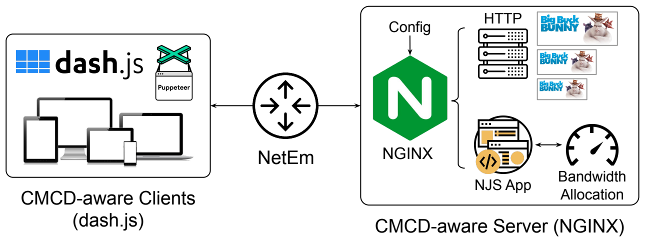

Figure 2: The implemented CMCD-aware system.

In a nutshell, the CMCD workflow is as follows:

(1) The CmcdModel.js class registers itself to call-back events from various auxiliary classes such as (1a) manifest loaded, (1b) playback rate changed, (1c) buffer level changed and (1d) playback sought, and updates its internal state when it receives a valid payload from one of those events.
(2) The HTTPLoader.js class asks the CmcdModel.js class to generate the CMCD query string, which in turn (2a) obtains the CMCD parameter values from its internal state or, if not present, the auxiliary classes. The set of CMCD parameters is assigned based on the object type to be requested (a manifest, media segment or initialization segment). The CmcdModel.js internally distinguishes different types of the objects since media segments may be augmented with additional CMCD parameters as needed. For example, for media segment requests, one may designate the segment type (*e.g.*, video, audio, *etc.*) and its duration.
(3) The CmcdModel.js class returns the CMCD query string to the HTTPLoader.js class.
(4) The HTTPLoader.js class then sends the HTTP GET request with the CMCD query argument for the requested object. Once the server receives the request, it performs the appropriate action(s) based on the CMCD parameter values, as exemplified in the next section.

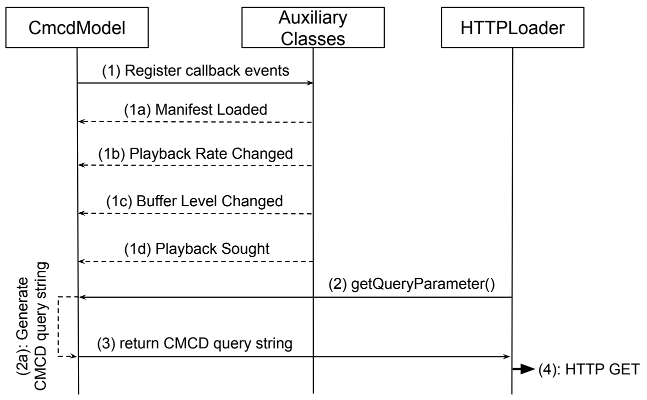
CmcdModel
Auxiliary Classes
HTTPLoader
(1) Register callback events
(1a) Manifest Loaded
(1b) Playback Rate Changed
(1c) Buffer Level Changed
(1d) Playback Sought
(2) getQueryParameter()
(2a): Generate CMCD query string
(3) return CMCD query string
(4): HTTP GET

Figure 3: The CMCD workflow in dash.js.

4.2 CMCD-Aware Server

We used NGINX [4] with the NGINX JavaScript (NJS) module to run the HTTP server and an NJS middleware application. NJS extends the NGINX configuration syntax to implement a basic application for bandwidth allocation and simplify the communication with the dash.js clients.

The CMCD-aware server's components are depicted in Figure 2. NGINX is acting as a frontend proxy server for the backend execution of the NJS application. NGINX also has a built-in HTTP server that stores the segments of different representations for various video content and their corresponding manifest files to serve the clients. The NJS application includes three main functions: *request processing and parsing*, *bandwidth allocation logic*, and *decision execution*. NGINX consists of modules that are controlled by directives specified in the configuration file (nginx.conf), which enables the HTTP server and NJS application (cmcd_njs.js) to leverage the required functionalities. Below, we provide details for the NJS application (cmcd_njs.js) functions:

(1) *Request processing and parsing*: This function processes each HTTP request received by NGINX with a URL path. It parses the query string to retrieve the values for the CMCD parameters and stores them in a JavaScript object (`cmcd_params`).
(2) *Bandwidth allocation logic*: This function is responsible for managing and dynamically allocating the bandwidth for each client with the aim of maintaining good viewer experience. It uses the received CMCD parameter values to make appropriate decisions. Inspired by the idea of a buffer-rate map function [26], we designed a simple and yet robust buffer-aware bandwidth allocation algorithm that finds a good relationship between the current buffer level and the bandwidth that should be allocated to a client. For this purpose, we use the following CMCD parameters: `buffer length` (bl), `max buffer` (com.example-bmx), `min buffer` (com.example-bmn), `buffer starvation` (bs) and `object type` (ot). Here, `max buffer` and `min buffer` are not from the CMCD specification but we added them to CmcdModel.js as they are an integral part to our buffer-aware bandwidth allocation algorithm given in Listing 1.
Our goal is to find a buffer length (bl)-to-rate (r) mapping function F(bl) $\longrightarrow r$ that can avoid (or at least reduce the number or duration of) rebufferings while not noticeably decreasing the

average video bitrate for each client. To achieve this goal, we define three situations for the client's buffer variation between the minimum (B_{min}) and maximum (B_{max}) levels: (S1) buffer underflow (bl < B_{min}), (S2) buffer overflow (bl > B_{max}) and (S3) buffer safe ($B_{min} \leq$ bl $\leq B_{max}$). The algorithm then allocates the bandwidth as follows:

(S1): The bandwidth allocation is fixed to the maximum rate capacity ($r = C_{max}$).

(S2): The bandwidth allocation is fixed to the minimum rate capacity ($r = C_{min}$).

(S3): The bandwidth allocation is computed using

$$r = C_{min} + ((1 - ((bl - B_{min})/B_{range})) \times C_{range}),$$

where $B_{range} = B_{max} - B_{min}$, $C_{range} = C_{max} - C_{min}$, $C_{max} = \alpha \times C$, $C_{min} = (1 - \alpha) \times C$ and C denotes the available total network capacity. Here, α is the bandwidth safety factor, which is set to 0.9 in our tests following the dash.js implementation [20].

```
function bufferAwareBandwidthAllocation(req) {
    var cmcd_params = processQueryArgs(req);
    var r = 0;
    var C = getAvailibleTotalCapacity();
    if (!('bl' in cmcd_params) || !('com.example-bmn'
        in cmcd_params) || !('com.example-bmx' in
        cmcd_params) || !('ot' in cmcd_params)) {
        return 0;  /* Disable bandwidth allocation */
    }
    if (cmcd_params['ot'] != 'v' && cmcd_params['ot']
        != 'av') { /* not video object */
        return 0;  /* Disable bandwidth allocation */
    }
    /* Buffer-to-rate mapping */
    var C_min = C x (1 - α);
    var C_max = C x α;
    var B_min = cmcd_params['com.example-bmn'];
    var B_max = cmcd_params['com.example-bmx'];
    var Bufferlength = cmcd_params['bl'];
    /* Case S1 */
    if (Bufferlength < B_min || ('bs' in cmcd_params)){
        r = C_max;
    }
    /* Case S2 */
    else if (Bufferlength > B_max) {
        r = C_min;
    }
    /* Case S3 */
    else {
        var B_range =  B_max - B_min;
        var C_range = C_max - C_min;
        r = ((1 - ((Bufferlength - B_min) / B_range)) *
            C_range) + C_min;
    }
    return r;
}
```

Listing 1: Buffer-aware bandwidth allocation algorithm.

Note that the bandwidth allocation logic may differ depending on the implementation choices, streaming scenarios under consideration, desired objectives and available CMCD parameters. The main objective of our bandwidth allocation algorithm is to reduce the impact of the rebuffering events, which is the most influential factor in maintaining a good viewer experience [41]. Thus, our algorithm allocates more bandwidth to the requests from the clients that will more likely experience a rebuffering.

(3) *Decision execution*: This function is responsible for applying the given bandwidth allocation decisions to the corresponding client requests. As configured in nginx.conf, it uses the `limit_rate` directive of the `http` module for per-request bandwidth allocation in NGINX.

An alternative approach to our bandwidth allocation logic is to have the client compute an appropriate value and send it using the `requested max. throughput` (rtp) parameter. This way, the server skips the computation part and only uses the `limit_rate` directive, which means less load on the server. However, in this case, the client does not know the value of C and as this value plays a critical role in the bandwidth allocation logic, we expect this alternative approach to perform worse than its server-side counterpart.

5 PERFORMANCE EVALUATION

5.1 Scenarios and Setup

To evaluate the CMCD-aware system, we created two scenarios: (*i*) an access link with 5–10 and (*ii*) an aggregation link with 20–30 concurrent streaming sessions, and we present our findings from the following three cases using these scenarios:

- Case 1: Scenario (*i*) with on-demand video sessions,
- Case 2: Scenario (*ii*) with on-demand video sessions, and
- Case 3: Scenario (*i*) with low-latency live video sessions.

Our setup is shown in Figure 2. To run the tests, we used one physical machine running Ubuntu 18.04.5 LTS with dual 20-core Intel E5-2630 v4 @ 2.20GHz processors and 192 GB memory. We ran the CMCD-aware dash.js (v3.1.3) clients on a Google Chrome browser (v88) with headless mode enabled using Puppeteer[1], which ran on top of Node.js. We used the default ABR scheme of dash.js, termed Dynamic (throughput-based + buffer-based heuristics). To emulate a realistic network based on our scenarios, we used *tc* NetEm at the server to throttle the total bandwidth available to the clients according to the bandwidth profiles *Cascade* and *Spike* defined by DASH-IF [41] and described in Table 2. We varied the bandwidth every 30 seconds depending on the scenario and looped the profile throughout the test, which was equivalent to the video duration.

Table 2: Bandwidth profiles used in the tests.

	Profile Name	Values (Mbps)	# of Clients
Access Link	CascadeX5	50, 20, 10, 5, 10, 20	5
	SpikeX5	50, 10	
	CascadeX10	100, 40, 20, 10, 20, 40	10
	SpikeX10	100, 20	
Aggregation Link	CascadeX20	200, 80, 40, 20, 40, 80	20
	SpikeX20	200, 40	
	CascadeX30	300, 120, 60, 30, 60, 120	30
	SpikeX30	300, 60	

On the HTTP server, we used two on-demand video datasets that were created based on Akamai's encoding recommendations [6].

[1]https://pptr.dev/

Both datasets were generated using the H.264 codec at 30 fps. Every video was chopped into segments of four seconds each (τ = 4 s). The first dataset (#1) consisted of a 10-minute animation video (*Big Buck Bunny*) and an ABR ladder of five representations: {180p@0.4 Mbps, 360p@0.8 Mbps, 432p@1.5 Mbps, 576p@2.5 Mbps, 720p@4.0 Mbps}. The second dataset (#2) had four four-minute long videos that covered a wide range of categories including sports (v1), movie (v2), animation (v3) and gaming (v4). These videos were downloaded from YouTube at the highest possible quality using youtube-dl[2] and encoded at 144p, 240p, 360p, 480p, 720p and 1080p, respectively at the six bitrates below:

- v1: {0.5, 0.7, 1.0, 1.5, 3.5, 5.0} Mbps,
- v2: {0.7, 1.0, 1.5, 2.0, 3.0, 6.0} Mbps,
- v3: {0.4, 0.7, 1.0, 1.5, 2.5, 4.0} Mbps,
- v4: {0.2, 0.5, 0.7, 1.0, 3.5, 4.5} Mbps.

To run Case 3, we incorporated the chunked encoding setup given in Twitch's grand challenge on the topic [41]. Specifically, we used (*i*) an FFmpeg live encoder to generate chunks of one frame (33 milliseconds at 30 fps), segments of one second and using an animation video (*Big Buck Bunny*) with an ABR ladder of {360p@0.2 Mbps, 480p@0.6 Mbps, 720p@1.0 Mbps}, (*ii*) an origin server that pushes available chunks to the clients using HTTP/1.1 chunked transfer encoding (CTE), and (*iii*) a dash.js client with the low-latency flag enabled and target latency set to three seconds.

In the clients and NJS application, we investigated two buffer configurations for Cases 1 and 2: (*a*) [$B_{min} = \tau \mid B_{max} = \tau \times \beta$] and (*b*) [$B_{min} = \tau \times 3 \mid B_{max} = \tau \times \beta \times 3$] where we set $\beta = 2$, and one for Case 3: (*c*) [$B_{min} = 1 \mid B_{max}$ = target latency = 3 s].

5.2 Results and Analysis

The primary goal of the three cases is to show the benefits of reducing rebuffering rate and duration without sacrificing video quality, when multiple clients compete for the available bandwidth. In each case, we repeated the tests five times and the tests were repeated for each bandwidth profile. We compared using CMCD with buffer-aware bandwidth allocation with not using CMCD (*i.e.*, running ABR only) using the following metrics:

- Avg. BR: Average bitrate across all clients (Mbps).
- Min. BR: Average bitrate for the client that consumed the lowest average bitrate (Mbps).
- Avg. RD: Average total rebuffering duration across all clients (s).
- Max. RD: Total rebuffering duration for the client that suffered from the longest rebuffering duration (s).
- Avg. RC: Average rebuffering count across all clients.
- Avg. SC: Average bitrate switching count across all clients.
- Avg. LL: Average live latency across all clients (s), Case 3 only.
- Max. LL: Average live latency for the client that experienced the longest live latency (s), Case 3 only.

5.2.1 Case 1: Access Link with Video-on-Demand Sessions. In this case, the goal is to show the benefits of using CMCD when ten clients concurrently run video-on-demand sessions on an access link. The test used the animation video from dataset (#1). The results for buffer configuration (*a*) are highlighted in Table 3.

[2]https://youtube-dl.org/

Table 3: Case 1 with dataset (#1) and buffer configuration (*a*).

	CMCD	NO CMCD	CMCD	NO CMCD
	CascadeX10		SpikeX10	
Avg. BR	3.13	3.33	2.61	3.20
Min. BR	2.90	3.12	2.30	2.68
Avg. RD	5.36	20.84	12.43	71.90
Max. RD	10.72	38.84	18.49	83.54
Avg. RC	4.72	11.04	8.68	25.48
Avg. SC	35.80	36.70	49.14	53.90

In Table 3, we see that enabling CMCD with buffer-aware bandwidth allocation significantly reduces Avg. RD, Max. RD and Avg. RC for both bandwidth profiles. The reductions are as follows: Avg. RD by 74% and 83%, Max. RD by 72% and 78%, and Avg. RC by 57% and 66% for Cascade and Spike, respectively. At the same time, the reduction in Avg. BR is not negligible, but it is much less significant. The bandwidth allocation algorithm aims to apportion a fair share across all the clients (based on their reported buffer levels), and hence, implicitly controls the decisions taken by the client-side ABR scheme. Due to this algorithm, Avg. SC is also reduced. The results for buffer configuration (*b*) are similar to the ones of (*a*), although the percentage of improvement is smaller in this case since the larger playback buffer size provides more robustness to the clients and makes rebuffering events less likely.

5.2.2 Case 2: Aggregation Link with Video-on-Demand Sessions. The goal in this case is to show the benefits of using CMCD when a variable number (20–30) of concurrent clients stream an on-demand video and compete at an aggregation link for the available bandwidth. When the number of clients increases, the competition for bandwidth significantly intensifies, which impacts QoE of the clients negatively. The results for Case 2 are given in Tables 4 and 5.

The left and right sides of Table 4 present the results for 20 and 30 clients, respectively, for dataset (#1) and buffer configuration (*a*). Enabling CMCD provides a reduction in Avg. RD of 41% and 48% (15% and 1%), in Max. RD of 38% and 35% (17% and 0%), and in Avg. RC of 16% and 36% (6% and 1%) for 20 (30) clients using the Cascade and Spike bandwidth profiles, respectively. We observe that the benefits of buffer-aware bandwidth allocation start diminishing when the number of clients increases. This is likely due to the simplicity of our bandwidth allocation algorithm.

Table 5 shows trends similar to the ones of Table 4. That is, the CMCD-enabled system achieved a better rebuffering performance for both bandwidth profiles using dataset (#2) and buffer configuration (*a*) with one notable difference: the absolute values for Avg. RD, Max. RD and Avg. RC were lower in Table 5 (compared to the left side of Table 4) even when we accounted for the duration difference of the videos in dataset (#1) and (#2). This was likely because in dataset (#2), four different videos were streamed by a total of 20 clients as opposed to the same video being streamed by all 20 clients. In Table 5, the clients obtained a reduction in Avg. RD of 85% and 67%, in Max. RD of 76% and 55%, and in Avg. RC of 73% and 46% for Cascade and Spike, respectively. These improvements in the rebuffering statistics cost a drop in Avg. BR of 8.5% for the Cascade and 11% for the Spike bandwidth profile.

Table 4: Case 2 with dataset (#1) and buffer configuration (*a*). The number of clients is 20 and 30 on the left and right side of the table, respectively.

	CMCD	NO CMCD	CMCD	NO CMCD	CMCD	NO CMCD	CMCD	NO CMCD
	CascadeX20		SpikeX20		CascadeX30		SpikeX30	
Avg. BR	3.20	3.39	2.78	3.16	3.32	3.34	3.07	3.11
Min. BR	2.88	3.03	2.22	2.55	2.93	2.97	2.33	2.41
Avg. RD	14.42	24.58	32.14	61.87	31.57	37.00	47.99	48.39
Max. RD	27.57	44.43	51.13	78.13	59.86	71.70	70.66	70.21
Avg. RC	9.24	11.05	14.41	22.36	14.17	15.13	19.03	19.26
Avg. SC	38.10	34.84	42.78	50.36	35.86	35.54	46.73	46.81

Table 5: Case 2 with dataset (#2), buffer configuration (*a*) and number of clients of 20.

	CMCD	NO CMCD	CMCD	NO CMCD
	CascadeX20		SpikeX20	
Avg. BR	4.19	4.58	4.05	4.55
Min. BR	0.97	0.99	0.99	0.98
Avg. RD	2.01	13.50	4.15	12.60
Max. RD	8.69	36.99	8.69	19.47
Avg. RC	1.10	4.15	2.50	4.60
Avg. SC	8.05	8.10	6.10	6.15

Table 6: Case 3 with buffer configuration (*c*) and live sessions of four minutes.

	CMCD	NO CMCD	CMCD	NO CMCD
	CascadeX5		SpikeX5	
Avg. BR	0.44	0.21	0.21	0.20
Min. BR	0.37	0.21	0.20	0.20
Avg. RD	3.56	4.45	3.62	4.91
Max. RD	3.87	5.16	4.27	5.44
Avg. RC	10.50	16.50	13.25	16.75
Avg. SC	21.75	18.00	5.75	3.25
Avg. LL	2.34	2.75	2.28	2.31
Max. LL	2.91	3.38	2.30	3.35

5.2.3 Case 3: Access Link with Low-Latency Live Sessions. Low-latency live (LLL) streaming [23, 29] has emerged recently as an area of attention in the streaming field for many researchers. The goal of LLL streaming is to achieve an end-to-end latency of few seconds in contrast to the traditional counterparts that exhibit latencies longer than half a minute. Low latency is achievable thanks to two key technology enablers: the Common Media Application Format (CMAF) [2] and CTE (RFC 7230).

Many factors still impact the latency, and therefore, the QoE, as the live content has to be captured, encoded, packaged and transferred from a server to a client over unpredictable network conditions. In Case 3, the goal is to determine whether using CMCD reduces rebuffering duration and/or count, and helps the latency stay below the target value set by the application.

The results for Case 3 are given in Table 6, where we see that upon enabling CMCD, the clients achieved a reduction in Avg. RD of 20% and 26%, in Max. RD of 25% and 22%, and in Avg. RC of 36% and 21% for Cascade and Spike, respectively. Moreover, the CMCD-aware clients stayed below the target latency (three seconds) without any violations, whereas Max. LL surpassed the target value when CMCD was disabled. We also observed an increase in Avg. BR by 110% when CMCD was enabled for the Cascade bandwidth profile since in this case the clients stalled less, and consequently, requested more segments from a high bitrate representation. These preliminary results are promising and we plan on conducting further investigations into using CMCD in LLL scenarios.

6 CONCLUSIONS AND FUTURE DIRECTIONS

The CMCD specification has recently emerged as an active cooperation paradigm that allows adaptive streaming clients to convey various player and playback related information to CDN servers. This information is quite useful to the CDN providers in understanding the main causes of QoE degradation, troubleshooting as well as improving the entire media distribution pipeline.

In this paper, we developed a proof-of-concept system based on the CMCD specification. The main objective was to investigate its uses in improving the performance for concurrently streaming clients sharing the network bandwidth. To achieve this objective, we designed a buffer-aware bandwidth allocation algorithm that found a good mapping between the current client buffer level and the bandwidth to allocate in order to reduce the chances for a rebuffering or the rebuffering duration. We evaluated this algorithm in a number of scenarios. The results show the benefits of using CMCD in helping to reduce the rebuffering rate and duration without noticeably sacrificing the video quality.

We would like to continue exploring the use cases for CMCD. For example, sending device type (or screen size) information as part of the CMCD query string may allow the server to do a more appropriate bandwidth allocation as was previously proposed in [10]. We also would like to better understand the limitations of CMCD and how it will interact with the new Common Media Server Data (CMSD) work (scheduled to start in Apr. 2021). These efforts will provide significant insights and help us in the development of the newer versions of the CMCD and CMSD specifications.

ACKNOWLEDGMENTS

This work was supported in part by Singapore Ministry of Education Academic Research Fund Tier 2 under MOE's official grant number MOE2018-T2-1-103, and in part by grant 31T102-UPAR-1-2017 from UAE University.

REFERENCES

[1] CTA-5004: Web Application Video Ecosystem–Common Media Client Data. [Online] Available: https://cdn.cta.tech/cta/media/media/resources/standards/pdfs/cta-5004-final.pdf. Accessed on Feb. 20, 2021.

[2] ISO/IEC 23000-19:2020 Information technology – Multimedia application format (MPEG-A) – Part 19: Common media application format (CMAF) for segmented media. [Online] Available: https://www.iso.org/standard/79106.html. Accessed on Feb. 20, 2021.

[3] ISO/IEC 23009-5:2017 Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 5: Server and network assisted DASH (SAND). [Online] Available: https://www.iso.org/standard/69079.html. Accessed on Feb. 20, 2021.

[4] High Performance Load Balancer Web Server. [Online] Available: https://www.nginx.com/, 2020.

[5] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery. In *IEEE INFOCOM*, 2012.

[6] Akamai. The Guide to Best Practices in Premium Online Video Streaming. In *White paper*, 2020.

[7] S. Akhshabi, L. Anantakrishnan, A. C. Begen, and C. Dovrolis. What happens when HTTP adaptive streaming players compete for bandwidth? In *ACM NOSSDAV*, 2012.

[8] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen. Server-based traffic shaping for stabilizing oscillating adaptive streaming players. In *ACM NOSSDAV*, 2013.

[9] A. A. Barakabitze, L. Sun, I.-H. Mkwawa, and E. Ifeachor. A Novel QoE-centric SDN-based Multipath Routing Approach for Multimedia Services over 5G Networks. In *IEEE ICC*, 2018.

[10] A. C. Begen. Spending quality time with the web video. *IEEE Internet Comput.*, 20(6):42–48, Nov./Dec. 2016.

[11] A. Bentaleb, A. C. Begen, S. Harous, and R. Zimmermann. Want to play DASH? a game theoretic approach for adaptive streaming over HTTP. In *ACM MMSys*, 2018.

[12] A. Bentaleb, A. C. Begen, and R. Zimmermann. SDNDASH: Improving QoE of HTTP adaptive streaming using software defined networking. In *ACM Multimedia*, 2016.

[13] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer, and R. Zimmermann. A survey on bitrate adaptation schemes for streaming media over HTTP. *IEEE Communications Surveys & Tutorials*, 21(1):562–585, 2019.

[14] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann. Bandwidth Prediction in Low-Latency Chunked Streaming. In *ACM NOSSDAV*, 2019.

[15] A. Bentaleb, P. K. Yadav, W. T. Ooi, and R. Zimmermann. DQ-DASH: A Queuing Theory Approach to Distributed Adaptive Video Streaming. *ACM TOMM*, 16(1), 2020.

[16] D. Bhat, A. Rizk, M. Zink, and R. Steinmetz. Network assisted content distribution for adaptive bitrate video streaming. In *ACM MMSys*, 2017.

[17] N. Bouten, M. Claeys, B. Van Poecke, S. Latré, and F. De Turck. Dynamic Server Selection Strategy for Multi-server HTTP Adaptive Streaming Services. In *IEEE CNSM*, 2016.

[18] N. Bouten, R. d. O. Schmidt, J. Famaey, S. Latré, A. Pras, and F. De Turck. QoE-driven In-network Optimization for Adaptive Video Streaming based on Packet Sampling Measurements. *Elsevier Computer Networks*, 81:96–115, 2015.

[19] J. Bruneau-Queyreix, M. Lacaud, and D. Negru. A Multiple-source Adaptive Streaming Solution Enhancing Consumer's Perceived Quality. In *IEEE CCNC*, 2017.

[20] DASH-IF. DASH Reference Client. [Online] Available: https://reference.dashif.org/dash.js/. Accessed on Feb. 20, 2021.

[21] L. De Cicco, S. Mascolo, and V. Palmisano. Feedback Control for Adaptive Live Video Streaming. In *ACM MMSys*, 2011.

[22] A. Detti, B. Ricci, and N. Blefari-Melazzi. Tracker-assisted Rate Adaptation for MPEG DASH Live Streaming. In *IEEE INFOCOM*, 2016.

[23] K. Durak, M. N. Akcay, Y. K. Erinc, B. Pekel, and A. C. Begen. Evaluating the performance of Apple's low-latency HLS. In *IEEE MMSP*, 2020.

[24] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-scale Control Plane for Video Quality Optimization. In *USENIX NSDI*, 2015.

[25] R. Houdaille and S. Gouache. Shaping HTTP Adaptive Streams for a Better User Experience. In *ACM MMSys*, 2012.

[26] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A Buffer-based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *ACM SIGCOMM*, 2014.

[27] J. Jiang, V. Sekar, I. Stoica, and H. Zhang. Shedding Light on the Structure of Internet Video Quality Problems in the Wild. In *ACM CoNEXT*, 2013.

[28] J. Jiang, S. Sun, V. Sekar, and H. Zhang. Pytheas: Enabling Data-driven Quality of Experience Optimization using Group-based Exploration-exploitation. In *USENIX NSDI*, 2017.

[29] W. L. Ultra-Low-Latency Streaming Using Chunked-Encoded and Chunked-Transferred CMAF. Akamai White paper. Online; accessed 10 January 2019.

[30] D. H. Lee, C. Dovrolis, and A. C. Begen. Caching in HTTP adaptive streaming: friend or foe? In *ACM NOSSDAV*, 2014.

[31] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran. Probe and Adapt: Rate Adaptation for HTTP Video Streaming at Scale. *IEEE Jour. Selected Areas Comm.*, 32(4):719–733, 2014.

[32] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang. A Case for a Coordinated Internet Video Control Plane. In *ACM SIGCOMM*, 2012.

[33] H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM*, 2017.

[34] A. Mehrabi, M. Siekkinen, and A. Ylä-Jääski. Joint Optimization of QoE and Fairness Through Network Assisted Adaptive Mobile Video Streaming. In *IEEE WiMob*, 2017.

[35] M. Mu, M. Broadbent, A. Farshad, N. Hart, D. Hutchison, Q. Ni, and N. Race. A Scalable User Fairness Model for Adaptive Video Streaming over SDN-assisted Future Networks. *IEEE Jour. Selected Areas Comm.*, 34(8):2168–2184, 2016.

[36] V. Nathan, V. Sivaraman, R. Addanki, M. Khani, P. Goyal, and M. Alizadeh. End-to-End Transport for Video QoE Fairness. In *ACM SIGCOMM*, 2019.

[37] NUS-OzU. CMCD-aware System. [Online] Available: https://github.com/NUStreaming/CMCD-DASH. Accessed on Feb. 20, 2021.

[38] S. Pham, P. Heeren, C. Schmidt, D. Silhavy, and S. Arbanowski. Evaluation of shared resource allocation using SAND for ABR streaming. *ACM TOMM*, 16(2s):1––18, 2020.

[39] Y. Qin, S. Hao, K. R. Pattipati, F. Qian, S. Sen, B. Wang, and C. Yue. ABR Streaming of VBR-encoded Videos: Characterization, Challenges, and Solutions. In *ACM CoNEXT*, 2018.

[40] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman. BOLA: Near-optimal Bitrate Adaptation for Online Videos. *IEEE/ACM Trans. Networking*, 28(4):1698–1711, 2020.

[41] Twitch. Grand Challenge on Adaptation Algorithms for Near-Second Latency. In *ACM MMSys*, 2020.

[42] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A Control-theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *ACM SIGCOMM*, 2015.

7 APPENDIX

The implementation of the CMCD-DASH system can be found in [37] (see release-v1.0).

7.1 Prerequisite Software

The CMCD-DASH system consists of a client-side and a server-side component, both of which were tested on a physical machine that ran Ubuntu 18.04.5. The following needs to be installed for the setup and execution:

- Ubuntu 18.04.5: https://www.ubuntu.com/download/desktop
- Node.js: https://nodejs.org/en/
- Grunt: https://gruntjs.com/
- Google Chrome: https://www.google.com/chrome/
- dash.js (v3.1.3): https://github.com/Dash-Industry-Forum/dash.js
- NGINX (v1.18): http://nginx.org/en/download.html
- tc-NetEm: https://wiki.linuxfoundation.org/networking/netem
- FFmpeg: https://www.ffmpeg.org/download.html
- Git: https://git-scm.com/downloads
- jq: https://stedolan.github.io/jq/

The user can download the prerequisites directly from the URLs above or use the Ubuntu package manager to install them, *i.e.*, (*i*) `sudo apt-get update`, (*ii*) `sudo apt-get install ⟨package_name⟩`.

7.2 Quick Setup

Follow the steps below for a quick setup and testing:

(1) Download the CMCD-DASH system from [37] using `git clone https://github.com/NUStreaming/CMCD-DASH.git`
(2) Run the NGINX server:
  - Navigate to the `cmcd-server/` folder.
  - Install the NJS module in NGINX using `sudo apt install nginx-module-njs`.
  - Open nginx/config/nginx.conf and edit `<PATH_TO_CMCD-DASH>` (under "`location /media/vod`") to indicate the absolute path to this repository.
  - Launch NGINX using `sudo nginx -c <PATH_TO_CMCD-DASH>/`cmcd-server/nginx/config/nginx.conf (note that the absolute path must be used).
  - Reload NGINX using `sudo nginx -c <PATH_TO_CMCD-DASH>/`cmcd-server/nginx/config/nginx.conf -s reload, if the configuration has changed.
  - Test the NJS application cmcd_njs.js with CMCD using `http://⟨MachineIP_ADDRESS⟩:8080/cmcd-njs/testProcessQuery?CMCD=bl%3D21300` and verify that it returns a value of 21300 for `buffer length` (bl).
(3) Run the dash.js client:
  - Navigate to the `dash.js/` folder.
  - Install the dependencies using `npm install`.
  - Build, watch file changes and launch samples page using `grunt dev`.
  - Test the dash.js application by navigating to `http://⟨MachineIP_ADDRESS⟩:3000/samples/cmcd-dash/index.html` to view the CMCD-enabled player.
(4) Run the experiment:
  - Navigate to the `dash-test/` folder.
  - Install the dependencies using `npm install`.
  - Edit `network_profile` in dash-test/package.json to specify the desired bandwidth profile for the test. The list of available bandwidth profiles are given in `dash-test/tc-network-profiles/`.
  - Edit `maxCapacityBitsPerS` in `cmcd-server/nginx/cmcd_njs.js` according to the selected bandwidth profile. Reload the NGINX config since we made a configuration change.
  - Edit `client_profile` in dash-test/package.json to specify the desired client profile (with CMCD or NO CMCD). There are two client profiles:
    - client_profile_join_test_with_cmcd.js
    - client_profile_join_test_no_cmcd.js
  - Update the setup parameters in the two client profile files based on the target scenario, such as the number of clients (`numClient`), minimum buffer (`minBufferGlobal`), maximum buffer (`maxBufferGlobal`), video location (`url`) and segment duration (`segmentDuration`). The set of video datasets are located in `cmcd-server/nginx/media/vod/`.
  - Start a test using `npm run test-multiple-clients`. Note that testing is done in Chrome headless mode by default.
  - Alternatively, to do a batch test with consecutive repeated runs for CMCD and NO CMCD (*e.g.*, a batch test of five CMCD and five NO CMCD runs), update the parameters in the two client profile files and `batch_test.sh`, and then run the batch test script with `sudo bash batch_test.sh`.
    - Note that the parameter values in batch_test.sh will overwrite those in package.json, hence, there is no need to edit the latter for the batch test run.
    - Note that the `jq` tool must be installed to use the batch test script.
    - If the batch test script is terminated prematurely, the background Chrome processes need to be killed.
  - Once the runs are finished, clear any previous tc setup using `sudo bash tc-network-profiles/kill.sh` (this must be run before starting any new run).
  - On completing the test run, results are generated in the `results/<timestamp>_multiple_clients/` folder ordered by the test run's timestamp.
  - To generate summary results across all clients in a test run, first navigate to the `results/` folder and then run `python generate_summary.py`.

Refer to the readme.md file in the GitHub repository [37] for troubleshooting of common issues.