EVALUATING THE PERFORMANCE OF LOW-LATENCY HTTP LIVE STREAMING

A Thesis

by

Kerem Durak

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in the
Department of Computer Engineering

Özyeğin University
August 2020

EVALUATING THE PERFORMANCE OF LOW-LATENCY HTTP LIVE STREAMING

Approved by:

Assoc. Prof. Ali C. Beğen, Advisor
Department of Computer Engineering
Özyeğin University

Asst. Prof. İsmail Arı
Department of Computer Engineering
Özyeğin University

Assoc. Prof. Müge Sayıt
Department of Computer Engineering
Ege University

Date Approved: 26 August 2020

To my wife Gülnur

ABSTRACT

In its annual developers conference in June 2019, Apple has announced a backwards-compatible extension to its popular HTTP Live Streaming (HLS) protocol to enable low-latency live streaming. This extension offers new features such as the ability to generate partial segments, use playlist delta updates, block playlist reload and provide rendition reports. Compared to the traditional HLS, these features require new capabilities on the origin servers and the caches inside a content delivery network. While HLS has been known to perform great at scale, its low-latency extension is likely to consume considerable server and network resources, and this may raise concerns about its scalability. In this thesis, we make the first attempt to understand how this new extension works and performs. We also provide a 1:1 comparison against the low-latency DASH approach, which is the competing low-latency solution developed as an open standard.

ÖZETÇE

Apple, 2019'daki geliştirici konferansında popüler HTTP canlı yayın protokolü olan HLS (HTTP Live Streaming)'e düşük gecikmeli canlı yayın yapmayı sağlayan ve buna rağmen ölçeklenebilir ve mevcut HLS istemcilerinde geriye dönük uyumlu olan yeni bir özellik geliştireceğini duyurdu. Bu özellik segment bölümleri oluşturmak, playlist delta güncellemeleri yapabilmek, playlist güncellemesini engellemek, rendition raporları oluşturmak gibi kabiliyetleri kapsıyor. Geleneksel HLS'e kıyasla bu kabiliyetler sunucu ve içerik dağıtım ağındaki önbelleklerde yeni geliştirmeler gerektiriyor. Geleneksel HLS'in büyük ölçeklerde çok başarılı olmasına karşın düşük gecikmeli versiyonunun önemli oranda sunucu ve ağ kaynaklarını tüketmesi olasıdır. Bu da onun ölçeklenebilir olmasıyla ilgili endişeler oluşturmaktadır. Bu tezde, düşük gecikmeli HTTP canlı yayın protokolü olarak duyurulan LLHLS (Low-Latency HLS)'in kabiliyetleri ve nasıl çalıştığı incelendi. Ayrıca açık kaynak olarak geliştirilmiş bir düşük gecikmeli canlı yayın çözümü olan Low Latency DASH ile birebir karşılaştırması yapıldı.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my thesis advisor Dr. Ali Cengiz Begen. He had always made time for me whenever I needed his guidance, he encouraged me and supported me along this process.

I would like to also thank Necmettin Akcay for his help and technical guidance along this way.

I would like to acknowledge my friends Yiğit and Boran who helped me during this study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Over-the-top (OTT) services delivering TV and on-demand video that use HTTP adaptive streaming (HAS) have become very popular globally in the last decade. The HAS systems scale well through the use of widespread content delivery networks (CDN), and HAS often delivers an uninterrupted and high-quality viewing experience to a variety of devices over a variety of networks as long as the amount of media buffered on the client side is large enough to absorb transient bandwidth problems. However, as the OTT services started offering more profitable, and also more latency-sensitive, content such as live sports, it became evident that the traditional HAS approach and client-side rate adaptation algorithms had to be tweaked for low-latency operations. To be on par with the ordinary (satellite/cable/over-the-air) TV broadcast, OTT services had to bring down the latency from the 30-60 second range to the 2-5 second range. Other services such as social media and game streaming have also become keen in low-latency offering as social media users and game streamers sought more effective ways to engage with their friends and audience, respectively.

1.1 End-to-End Live Latency (aka Latency)

To understand how the latency could be lowered, we first need to understand what contributes to the latency. Latency is defined as the delta time between the capture and rendering, and includes the time spent for capturing, encoding, packaging, publishing to the origin server, distributing through the CDN, buffering at the client and decoding. Typically, HAS uses media segments of 2-10 seconds, and the packager, origin server, caches and client operate at the segment granularity. For example, a segment is not pushed to the origin server by the packager till it is fully ready and

the origin server does not serve it till it is fully available. At the client, the decoding and playback does not commence till several segments have been received. These serialized delays add up, and the resulting latency turns out to be three to six times the segment duration. Here, an obvious trick is to use a shorter segment duration. While this can effectively lower the latency, it comes at the expense of reducing the encoding efficiency and increasing the network traffic. A better approach is to decouple the segment duration from the latency.

1.2 The Two Saviors: CMAF and CTE

Back in 2016, Apple and Microsoft joined forces in MPEG to develop an extensible encoding standard to unify the streaming media format without mandating a specific manifest format or delivery protocol. A relatively fast standardization effort resulted in the Common Media Application Format (CMAF) [1]. CMAF uses ISO-BMFF (ISO/IEC 14496-12) for the common core media segments, and permits a choice of a DASH manifest (MPD), an HLS playlist or any other format for the manifest [2]. The ultimate motivation behind CMAF was to eliminate the proliferation of alternatively packaged content in use, and today we witness that the industry has been making a good effort in this direction.

CMAF media objects are derived from ISO-BMFF using movie fragments. The smallest CMAF media object is a media sample, which is media data associated with a single decode start time and duration. The media samples in a CMAF fragment are constrained to be independently decodable. A CMAF fragment typically consists of one MovieFragmentBox (moof) and MediaDataBox (mdat) pair, but can also contain more than one of these pairs. If the CMAF fragment contains multiple such pairs, each pair is called a CMAF chunk and contains a consecutive subset of the CMAF fragment's media samples. CMAF chunks are addressable media objects, and are particularly critical in low-latency live streaming since a CMAF chunk can be sent

as soon as it is encoded and packaged without waiting for the encoding and packaging of the subsequent chunks in the same CMAF fragment (and the subsequent CMAF fragments constituting the respective CMAF segment). CMAF chunks can be temporarily stored, referenced by a URL in the manifest and delivered as objects. However, CMAF chunks are not necessarily self-decodable, which means not every chunk can be used as a switching point. CMAF chunks also cannot reduce the latency themselves. To reduce the latency, they need to be paired with an appropriate delivery method and this is where chunked transfer encoding (CTE) comes into play [3].

Suppose that the packager generates a CMAF chunk for every video frame, a CMAF fragment every two seconds and a CMAF segment every six seconds. From the packager, CMAF chunks can be transferred to the origin server using a reliable object delivery protocol. If the origin server is running HTTP/1.1 (RFC 7230), it can pull each CMAF chunk using CTE. Also suppose that the streaming client determines the live-edge segment using the information in the manifest, then makes an HTTP GET request for that segment. The origin server returns the requested segment, which could be still incomplete, via CTE, where each HTTP chunk contains a single CMAF chunk. However, note that the particular mapping between the CMAF and HTTP chunks is implementation specific. As CMAF chunks are received, the streaming client can consume them.

Figure 1 demonstrates a live streaming scenario where the content is segmented (each of four seconds) but not chunked. The numbers inside the boxes indicate the segment numbers. The streaming client joins the live session at 18 seconds into the content. If the streaming client is the native HLS client on iOS/iPadOS/tvOS, it will check the playlist and see that the segments #1 through #4 are available. It will then request segments #2, 3 and 4 since the native HLS client requires at least three segments in the buffer before the playback can start. Assuming the segment fetching

time is negligible (meaning that the round-trip time from the client to the server is small and the available bandwidth is plenty), the client can start the playback very quickly but with a latency of 14 seconds. If we required our client to have only one segment in the buffer before the playback started, we could reduce the latency to six seconds. If we programmed our client to achieve the lowest latency, it could prudently defer sending a request for segment #4 at the time of join, wait for two seconds, and then send a request for segment #5. This way, the latency drops to four seconds, although the startup delay increases by two seconds.

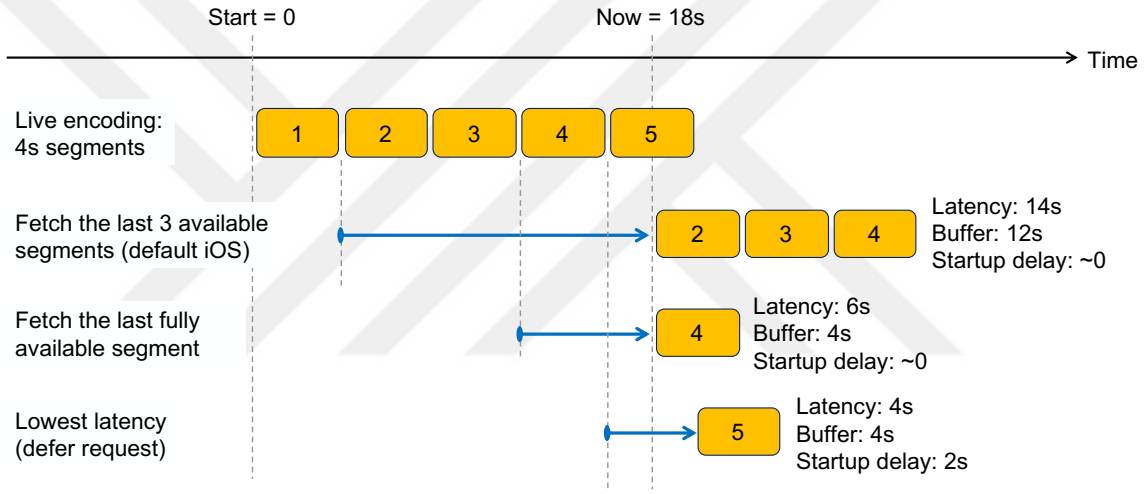

Figure 1: Different playback strategies for the segmented content (four second segments).

Figure 2 illustrates the same live streaming scenario with the chunked content (four second segments, one second chunks) and CTE enabled. The client joins the session at 18 seconds and determines the live-edge segment (segment #5). In the simplest fashion of using CTE, the client can commence the playback as soon as the first chunk (#5A) is received achieving a latency of two seconds. On the other hand, a more adventurous client can decode chunks #5A and #5B, but start the playback from #5B, which further reduces the latency to one second. As we can see from the illustration, the combination of CMAF and CTE can eliminate the serialized delays and deliver the media with lower latency. Note that upon join, if the client requested

an earlier segment that had been already fully available, it would likely receive the complete segment in a single response from a cache in the CDN and incur a higher latency, since CTE would not be useful in this case.

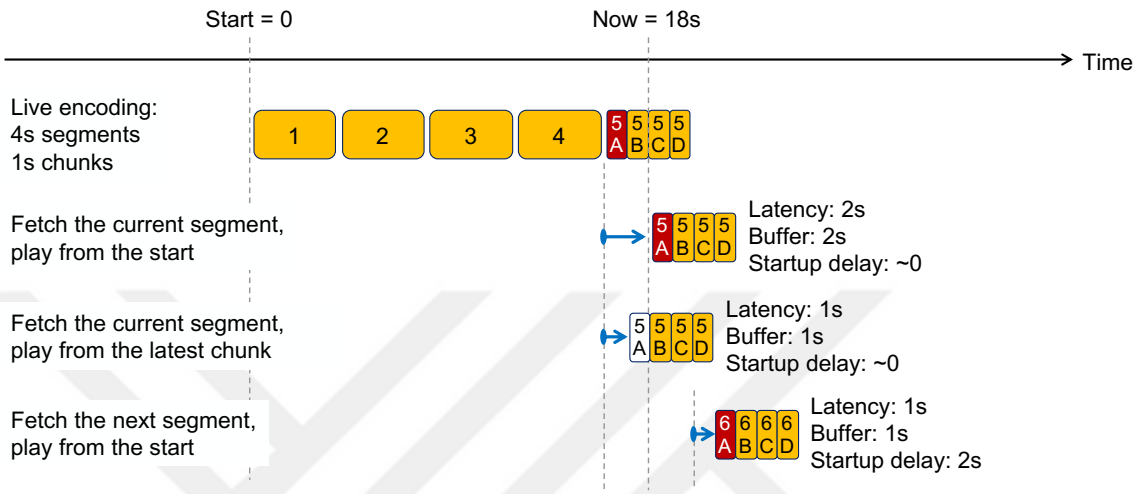

Figure 2: Different playback strategies for the chunked content (four second segments, one second chunks).

1.3 Industry Efforts

With the growing interest in low latency, the industry moved quickly and today we have three solutions. (*i*) DASH Low-Latency (DASH-LL) [4] was introduced by DASH-IF and DVB in 2019; (*ii*) Low-latency HTTP Live Streaming (LHLS) [5] was first introduced by Twitter's Periscope application in 2018 and then improved by Twitch in 2019; (*iii*) Low-Latency HTTP Live Streaming (LL-HLS) [6] was introduced by Apple in June 2019. At the high level, a common feature across all these solutions is that they require the media to be encoded and delivered in chunks. However, there are some obvious as well as subtle differences in their requirements and implementations. These are highlighted in Table 1.

1.4 The Focus of This Work

Low-latency streaming is a vast topic. The high-level architecture is easy to understand, however, there are several important details to pay attention to when it comes

Table 1: High-level comparison of DASH-LL, LHLS and LL-HLS.

	Chunked Content	Use of CTE	Describe Segment Structure	Per-chunk Manifest Refresh	Line-speed Delivery	Smart Origin to Modify Manifest	Deterministic Startup
DASH-LL	✓	✓	×	×	×	×	×
LHLS	✓	✓	×	×	×	×	×
LL-HLS	✓	×	✓	✓	✓	✓	✓

to a robust implementation. These details are documented for DASH-LL primarily in [4, 7, 8], for LHLS in [5] and for LL-HLS in [6, 9]. For DASH-LL, open-source client implementation [10] as well as encoder/packager software [11, 12] are also available. Independent of the technology, low-latency streaming brings up new challenges in measuring the available bandwidth accurately for proper rate adaptation. These challenges and a solution approach have been presented in [13, 14]. In this thesis, the goal is not to go into further details of the issues above, but rather to conduct the first performance evaluation of the LL-HLS solution. At the writing of this thesis, the LL-HLS specification has not been finalized and the test software from Apple has not left the beta stage, although three different betas had been released for iOS 13. This means that the stable version of the LL-HLS (which is expected in iOS 14 in late 2020) may show better performance than what we report in this study. Nevertheless, we believe our study is useful to document the evolution of LL-HLS.

In the rest of the thesis, Section 2 provides an overview for the LL-HLS protocol detailing the individual features that are most important in our study, Section 3 describes the test environment, Section 4 presents our findings and Section 5 provides the conclusions.

CHAPTER II

THE LL-HLS PROTOCOL EXTENSION

Apple's HLS (RFC 8216), which has been around for more than a decade, has been proven to be globally scalable to deliver live and on-demand content. While it is simple and works on virtually any type of device, it has failed in delivering content with low latency compared to competing solutions. To address this weakness, in June 2019, Apple has announced a backwards-compatible protocol extension [15], which added new functionalities to the traditional HLS such as generation of partial segments, playlist delta updates, blocking of playlist reload, preload hints and rendition reports[1].

During the announcement, the audience was surprised by Apple's decision to go with a non-CTE approach despite the popularity of the community-developed LHLS solution [5] at that time. To get more or less the same benefits, Apple instead opted for introducing a new concept called partial segments and using HTTP/2 (RFC 7540) with Server Push capabilities. This quickly made many developers and CDN providers skeptical since the LL-HLS extension introduced a number of new requirements for the origin servers and CDNs. For example, the origin servers must be able to modify the HLS playlists (based on the query arguments in the requests) and the caches in a CDN must implement certain new features including support for HTTP/2 and Server Push (more on this in Section 2.6). However, at the moment, with a few exceptions, none of the commercial CDNs support HTTP/2 and Server Push, and the required development is reportedly a big undertake for them. If the origin servers and/or the CDN caches cannot meet the LL-HLS requirements, LL-HLS will not perform as expected or simply fall back to the traditional HLS behavior, which means it can only

[1] `https://developer.apple.com/videos/play/wwdc2019/502/`

deliver content with a high latency.

LL-HLS still uses the playlist approach, which means that the playlists have to be updated more frequently and the clients have to request every partial segment separately to achieve low latency. Depending on the choice of partial segment sizes, the request rate of an LL-HLS client can be substantially higher as we need to consider the GET requests not only for the partial segments but also for the playlist updates. Considering that live streaming events draw a big audience, there will be many concurrent clients, each with a high request rate and this can be a major concern for the CDN and network service providers due to increased network traffic.

On the more positive side, LL-HLS playlists describe the internal structure of the segments. Using the `INDEPENDENT=YES` attribute, the partial segments containing an independent frame are indicated. This way, the LL-HLS client can determine the optimal partial segment to fetch upon starting up or switching between the renditions (alternative streams at different bitrates). LL-HLS clients also experience more deterministic startup delays. By using the `PART-HOLD-BACK=<seconds>` attribute in the playlist, they are told to start up that many seconds behind the live edge, unless the value specified by `AVPlayerItem.configuredTimeOffsetFromLive` is greater. The recommended value for `PART-HOLD-BACK` is at least three times the partial segment duration.

The primary documentation for the LL-HLS extension is currently [6], although we expect a more formal documentation through [16] at a later stage. As stated in [6], LL-HLS introduces the following main changes into HLS.

2.1 Partial Segments

LL-HLS divides segments into smaller pieces, called partial segments or parts in short, which are similar to CMAF chunks. Partial segments can be as short as a frame duration and as long as half the segment duration. Also, partial segments do

not have to have identical durations. As for CMAF chunks, partial segments are encoded, published and added to the media playlist earlier than the parent segment. For example, with a segment duration of four seconds and partial segment duration of one second, the first partial segment is published in one second followed by the three remaining partial segments, published one second apart from each other. In LL-HLS, the ads also need to be divided into and delivered as parts. Blocking of playlist reload and preload hints must be implemented for the ads just like the media content, although the origin serving the ads does not need to enforce the blocking semantics as ads are prerecorded.

A long-standing problem with HLS has been the playlist bloat. In a long-running session with a large timeshift window, the playlist becomes quite large in size. With partial segments listed, the playlist can grow even faster. To avoid this problem, LL-HLS requires the partial segments older than three segment (target) durations from the live edge to be removed from the media playlist. In their stead, only the full segments are listed in the media playlist.

2.2 Playlist Delta Updates

In LL-HLS, clients initially request the master playlist and the media playlists, and then request only a delta update for every part. A delta update contains only the last few segments along with the parts at the tail of the playlist. Delta updates are smaller in size, and hence, reduce the amount of traffic for playlist refreshes.

2.3 Blocking of Playlist Reload

LL-HLS introduces an ability on the server to block a playlist reload request from a client. When the client issues a GET for a media playlist update, it can add query parameters to specify that it wants the playlist response to include a future segment. It is the responsibility of the server to hold onto the request until a playlist that contains the requested segment is available. This feature is useful to eliminate

constant polling of the servers. Coalescing duplicate blocking requests also helps reduce the load on the CDNs.

2.4 Preload Hints

To eliminate the unnecessary round-trip delays due to premature requests, the EXT-X-PRELOAD-HINT[17] tag can be used in the playlist, which informs the client about the upcoming partial segments and media initialization sections. This way, the client can issue a GET request for a hinted resource and the server responds to the request as soon as the resource becomes available.

2.5 Rendition Reports

In LL-HLS, each media playlist contains information about the segments in other renditions (listed in the master playlist) to allow faster switching with minimum number of round trips. The rendition reports are carried in the `EXT-X-RENDITION-REPORT` tag and provide information such as the last media sequence number and part currently in the media playlist of that rendition.

2.6 Apple's LL-HLS Beta Tools

Apple has released beta tools for generating media and playlists for LL-HLS. The first version of the tool (referred to as v40) was published in September 2019. The tool included two executables for encoding and packaging purposes, and a PHP script that implemented the server-side LL-HLS features. The first executable, `tsrecompressor`, encodes a continuous stream of audio and video either programmatically (with an image derived from the system clock for latency measurement purposes) or by capturing input from the camera and microphone at the target bitrates. The encoded streams are multicast in MPEG2-TS format to local UDP ports, each of which is then packaged for LL-HLS by a separate instance of `mediastreamsegmenter`. `mediastreamsegmenter` outputs a live media playlist with its partial segments, which

are used by the origin server. The `lowLatencyHLS.php` script implements blocking of playlist reload, playlist delta updates and rendition reports.

In v40, partial segments were delivered via HTTP/2 Push in response to a blocking playlist request. The Push reduced the total time for the LL-HLS client to receive the partial segments. However, this feature has been a concern for CDNs, which are optimized for fast media delivery but not for playlist distribution, since playlists are often generated/modified by non-CDN entities for ad insertion and other purposes. With HTTP/2 Push, the CDNs would have to push out the playlists and segments from the same edge server.

The second version, called v64, came out in February 2020. The major change was that while HTTP/2 remained mandatory, the use of HTTP/2 Push was replaced with a new `EXT-X-PRELOAD-HINT` tag. With this tag, the origin server can inform the LL-HLS client about an upcoming part within the playlist. Then, the client can issue a speculative GET request for that part along with its playlist. When the part becomes available, the origin sends both the updated playlist and the part itself, saving the client a second round-trip time for the part request without requiring HTTP/2 Push. This change was widely welcome by the industry as CDNs could now deliver the segments without touching the playlists and they could easily cache the request without requiring an extra round-trip between the edge and the origin since request is initiated from the client. Yet, the impact of large amount of blocking connections still raises concerns, as CDNs are designed to serve requests as quickly as possible.

In v64, `ll-hls-origin-example.go` script was also added to implement the LL-HLS Origin API for LL-HLS media playlists (as an alternative to the `lowLatencyHLS.php` script). This Go script sets up LL-HLS directives and the web server altogether. The third (latest) version of the tool is v73 and came out in March 2020, and most likely included some bug fixes and performance improvements.

```
#EXTM3U
# Following the example above, this Playlist is a response to:
# GET https://example.com/2M/waitForMSN.php?_HLS_msn=273&_HLS_part=3 &_HLS_skip=YES
#EXT-X-TARGETDURATION:4
#EXT-X-VERSION:9
#EXT-X-SERVER-CONTROL:CAN-BLOCK-RELOAD=YES,PART-HOLD-BACK=1.0,CAN-SKIP-UNTIL=12.0
#EXT-X-PART-INF:PART-TARGET=0.33334
#EXT-X-MEDIA-SEQUENCE:266
#EXT-X-SKIP:SKIPPED-SEGMENTS=3
#EXTINF:4.00008, fileSequence269.mp4
#EXTINF:4.00008, fileSequence270.mp4
#EXT-X-PART:DURATION=0.33334,URI="filePart271.0.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart271.1.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart271.2.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart271.3.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart271.4.mp4",INDEPENDENT=YES
...
#EXT-X-PART:DURATION=0.33334,URI="filePart271.11.mp4"
#EXTINF:4.00008, fileSequence271.mp4
#EXT-X-PROGRAM-DATE-TIME:2019-02-14T02:14:00.106Z
#EXT-X-PART:DURATION=0.33334,URI="filePart272.a.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart272.b.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart272.c.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart272.d.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart272.e.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart272.f.mp4",INDEPENDENT=YES
...
#EXT-X-PART:DURATION=0.33334,URI="filePart272.l.mp4"
#EXTINF:4.00008, fileSequence272.mp4
#EXT-X-PART:DURATION=0.33334,URI="filePart273.0.mp4",INDEPENDENT=YES
#EXT-X-PART:DURATION=0.33334,URI="filePart273.1.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart273.2.mp4"
#EXT-X-PART:DURATION=0.33334,URI="filePart273.3.mp4"
#EXT-X-PRELOAD-HINT:TYPE=PART,URI="filePart273.4.mp4"

#EXT-X-RENDITION-REPORT:URI="../1M/waitForMSN.php",LAST-MSN=273,LAST-PART=3
#EXT-X-RENDITION-REPORT:URI="../4M/waitForMSN.php",LAST-MSN=273,LAST-PART=3
```

Figure 3: Example Low Latency HLS Playlist

Figure 3 shows an example LL-HLS playlist as a response to this request:

`https://example.com/2M/waitForMSN.php?_HLS_msn=273&_HLS_part=3&_HLS_skip=YES`

In this playlist, segments are 4 seconds and partial segments are 0.33 seconds long thus having 12 partial segment of a full segment. Only the last 3 segments' partial segments are delivered. Table 2 explains some of the new media playlist tags for Low-Latency HLS.

Table 2: LL-HLS Playlist Tags

EXT-X-PART	Indicates the partial segments in the media playlist
EXT-X-SKIP [18]	Number of skipped segments when server issues a playlist delta update
EXT-X-PART-INF	Provides information about partial segments
PART-TARGET	Indicates the duration of partial segments
EXT-X-RENDITION-REPORT	Carries a Rendition Report in the media playlist
EXT-X-SERVER-CONTROL	Indicates support for features such as Blocking Playlist Reload and Playlist Delta Updates
EXT-X-PRELOAD-HINT	Hints that a resource will be needed to play back an upcoming part

CHAPTER III

TEST ENVIRONMENT

We installed the beta tools provided by Apple on MacOS Catalina 10.15.3 to run as the server. Activating the low-latency mode required HTTP/2 and PHP support to be enabled on this server. To that effect, we used a Docker container [19] that ran an Nginx server instance (v1.16.0). In the beta stage, enabling the low-latency mode on the clients requires a native app (using the AVPlayer) on devices running iOS 13 and the app must carry the `com.apple.developer.coremedia.hls.low-latency` entitlement (which will not be required in iOS 14). In our testing, we used an iPhone 6S Plus with iOS 13.3.1 and developed an app starting from the HLS catalog application [20]. We connected the server and client to each other over the local area network, and emulated different types of connections between them, as needed.

In our testing, we were primarily interested in two metrics, startup delay and latency. Startup delay is the delta time from when the play button was clicked (or a particular stream was selected) to the time the playback started. Latency is the delta time from a frame's capture to its playback. Measuring latency required frame timestamps to be conveyed to the client as well as synchronizing the server and client's system clocks. For the former, we used `tsrecompressor`'s programmatic option to create a stream of images, each of which encoded the corresponding timestamp into a QR code. For the latter, we used a local SNTP server [21] that we ran on our server. The SNTP server synchronized its time from `time.apple.com` before each test. To eliminate time decay, we synchronized the client's clock with the server's, using Kronos[1], before obtaining each latency sample.

[1] `https://github.com/lyft/Kronos`

CHAPTER IV

RESULTS

We obtained the results by using different combinations of segment and partial segment durations, and network conditions. For reliable results, we repeated each test fifty times. In the box plots presented, boxes extend from the first quartile to the third quartile of the samples, with a line at the median. Vertical lines extending from the boxes are whiskers, indicating the range of the samples.

4.1 Startup Delay and Latency

We tested segment durations of one through six seconds. For each segment duration, we used five different partial segment durations, varying from one tenth to half of the segment duration. For example, for one-second segment duration, we used 100, 200, 300, 400 and 500 ms partial segments. Note that not each partial segment duration evenly divides the segment duration. In those cases, the last partial segment turned out to be shorter than the preceding ones.

Figure 4 shows the startup delay performance for a subset of the segment and partial segment duration combinations for the three versions. We see that the startup delays achieved by v64 and v73 are slightly higher than the ones achieved by v40 for shorter segment durations. For longer segment durations, v40 performs slightly worse than the other two. On the other hand, v73 experiences slightly higher startup delay variation, although such level of variations are not easily perceptible in practice. Figure 5 shows that the startup delay is also affected by the encoding bitrate.

On the latency side, Figure 6 shows that latency is affected by partial segment duration, while there is a little effect of segment duration on the latency (not shown in

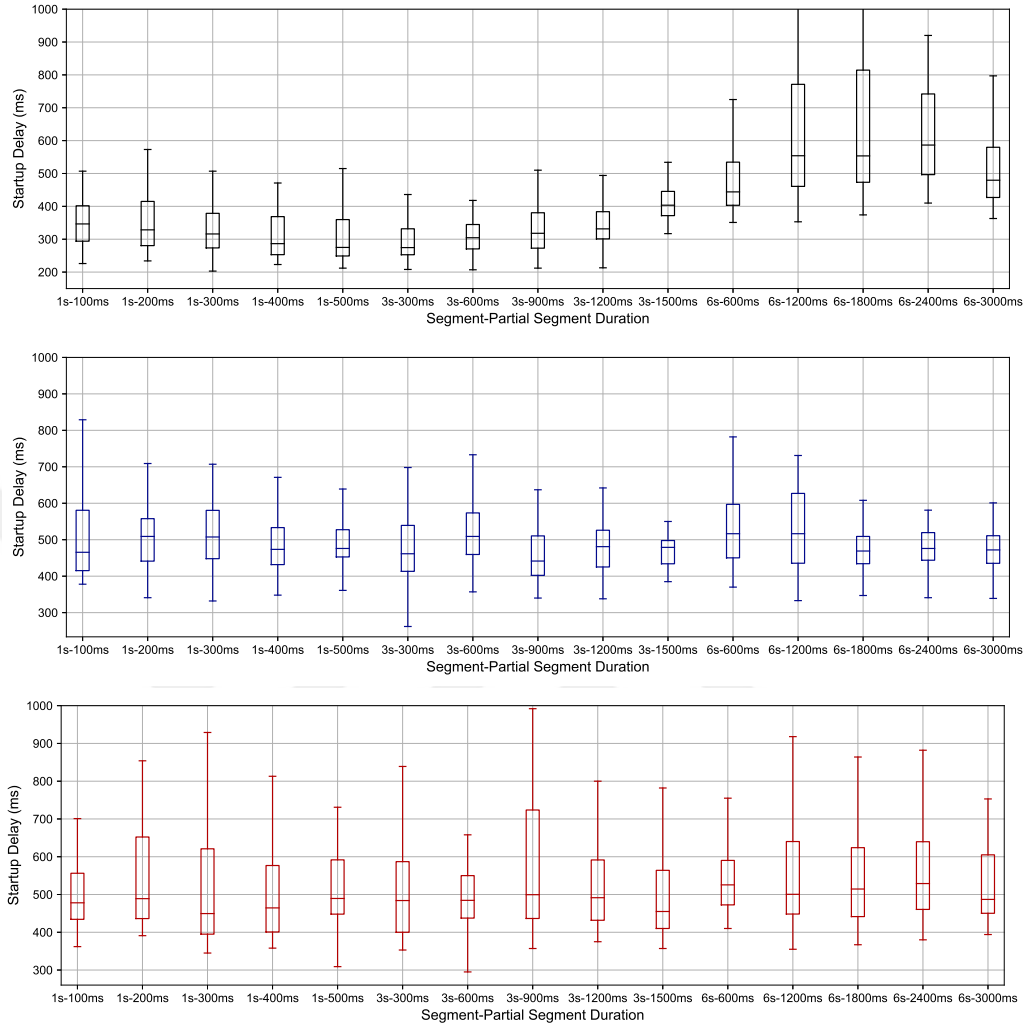

Figure 4: Startup delay comparison of v40 (top), v64 (middle) and v73 (bottom) for the 0.5 Mbps stream.

the figure). Best latency performance, which is around 1500 ms, is obtained with one-second segment duration and 10 partial segments. Examining the latencies achieved by different versions, we see no significant difference.

4.2 Network Traffic

In LL-HLS, network traffic (in terms of both the number of request/response exchanges and the number of total bytes sent) varies according to the segment and partial segment durations we pick. The number of requests for the playlist updates and partial segments substantially increases with shorter partial segment durations,

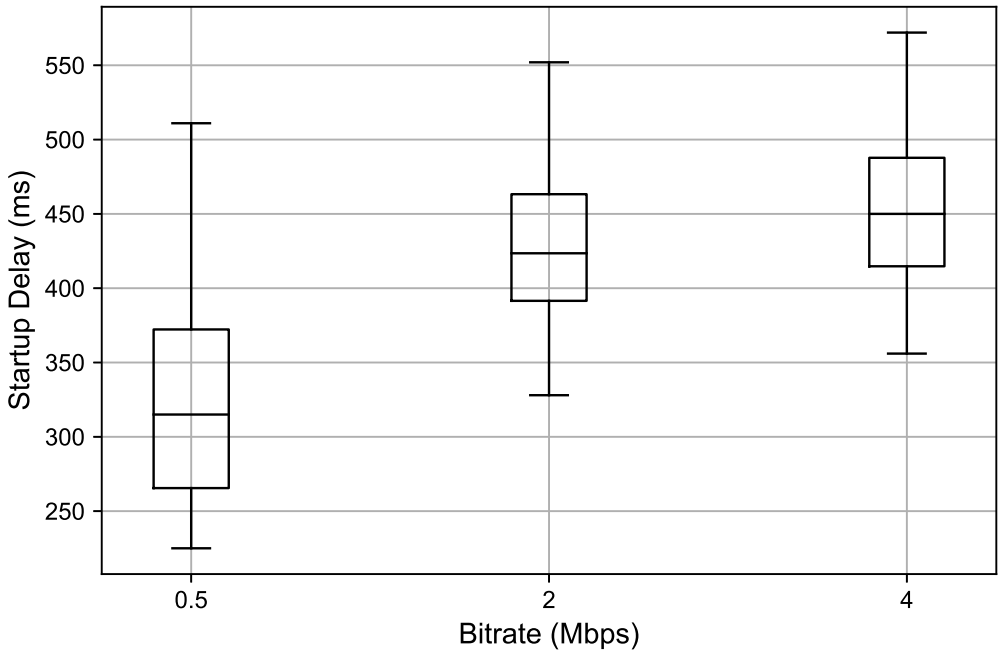

Figure 5: Startup delay for different bitrates: v40, segment duration of two seconds, partial segment duration of 400 ms.

and this has been a concern for CDN and network service providers since the introduction of LL-HLS.

In the next experiment, we compare the network traffic generated in a single LL-HLS (v64) client system against the traffic generated in a single dash.js client [10] system. For the latter, the DASH-LL stream was sourced from [22], which was encoded at 2 Mbps and 30 f/s. This stream had a segment duration of two seconds and each video frame was packaged into a single CMAF chunk, with a duration of 34 ms. On the other hand, we locally generated the LL-HLS stream at 2 Mbps and 30 f/s. The segment and part durations were set to two seconds and 34 ms, respectively. Over a period of 60 seconds, the dash.js client made 30 GET requests (for 30 segments only as no update for the template-based manifest was needed) and downloaded approximately 15 MB of media. In the same period of time, the LL-HLS client made approximately 1960 GET requests, about 1800 of which were sent for the partial segments and the remaining 160 were sent for the playlist updates. While

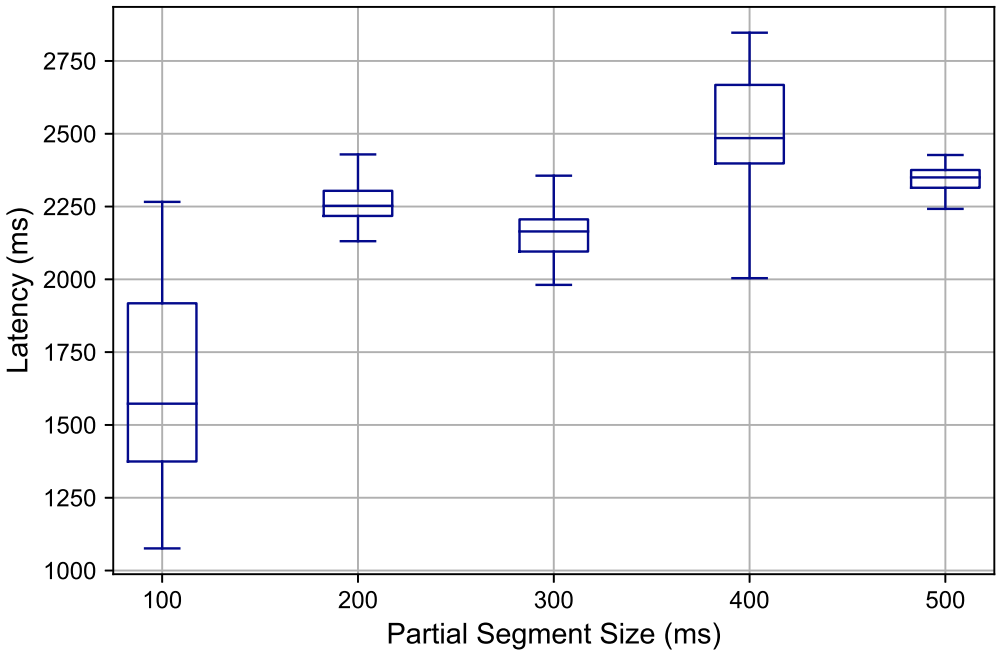

Figure 6: Latency for different partial segment durations: v64, segment duration of one second.

the overhead of additionally downloaded non-media data (the playlist updates) was about 1%, the ratio between the number of GET requests for DASH-LL and LL-HLS (1:65) was significant.

In practice, we do not expect parts or chunks to be this short since it does not bring notable advantages and in a practical deployment, the parts/chunks should be at least 100 ms in duration. While the number of GET requests does not depend on the chunk duration in DASH-LL, that number depends on the part duration in LL-HLS. For part durations of 100 ms, 200 ms and 500 ms, the experiment above resulted in about 600, 300 and 120 GET requests for the partial segments, and 335, 221 and 100 GET requests for the playlist updates, respectively.

Our preliminary investigation on CPU usage also showed that the HTTP server used approximately four times the cycles for 20 LL-HLS clients than it did for 20 dash.js clients. It is early to draw any conclusions from this finding before LL-HLS comes out of the beta and efficient implementations become available, however, the

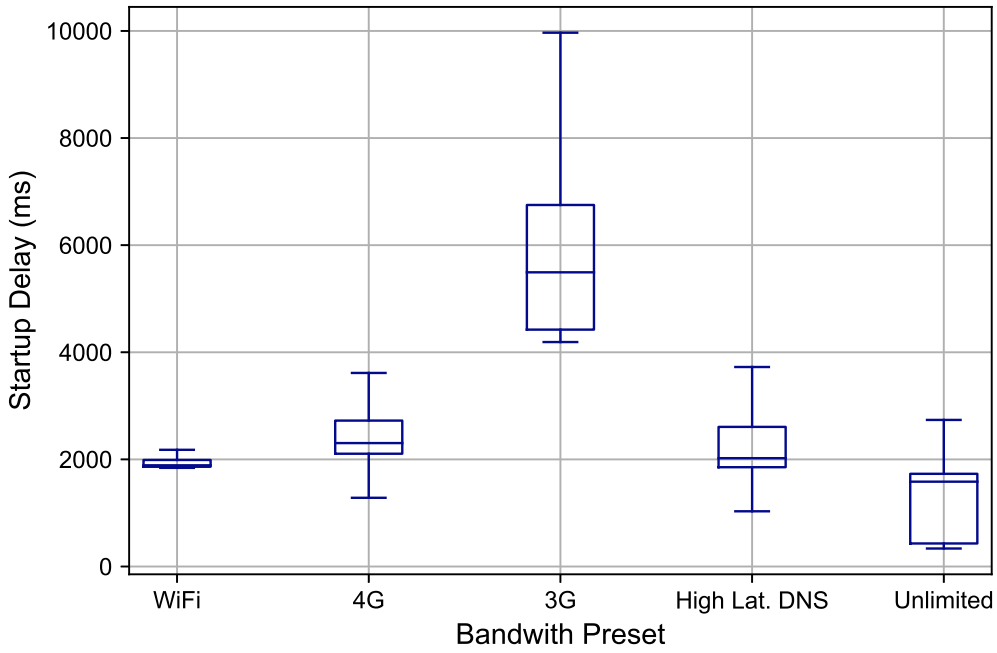

Figure 7: Startup delay comparison for different network conditions: v64, segment duration of four seconds, partial segment duration of one second.

increased number of GET requests is a peculiarity of LL-HLS and its impact should be investigated more thoroughly in the future.

4.3 Network Conditions

With v64, Apple has provided a sample LL-HLS stream[1]. We used this stream to compare the low-latency performance with different network conditions, which were emulated using Apple's Network Link Conditioner tool (included in Additional Tools for Xcode). Figures 7 and 8 show the startup delay and latency performances for WiFi, 4G, 3G, High Latency DNS and Unlimited presets.

[1] `https://ll-hls-test.apple.com/master.m3u8`

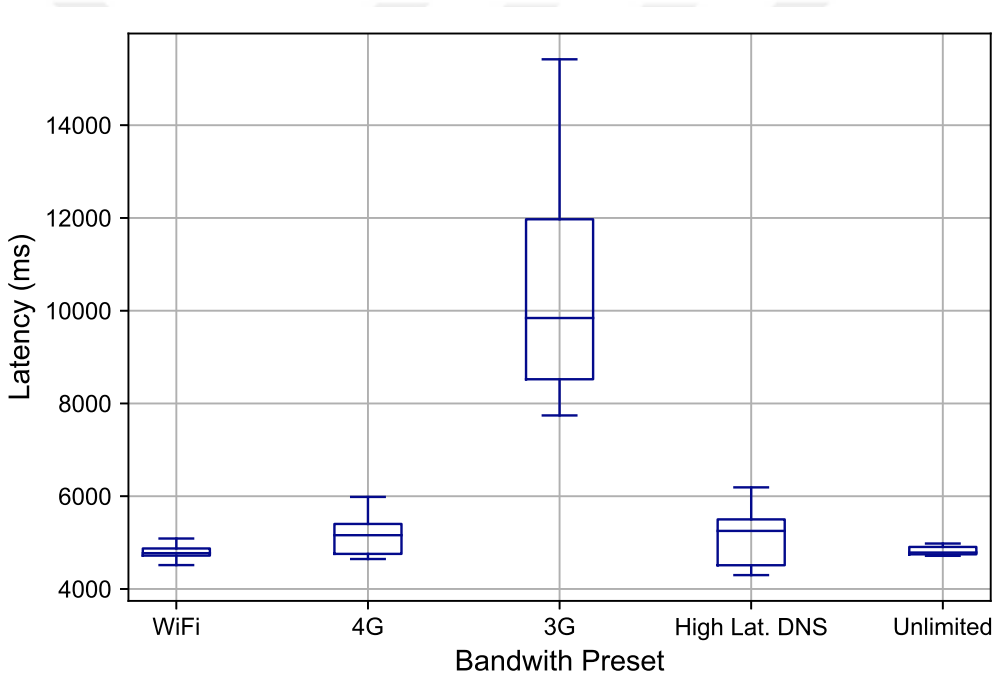

Figure 8: Latency comparison for different network conditions: v64, segment duration of four seconds, partial segment duration of one second.

CHAPTER V

CONCLUSION

In this thesis, we investigated Apple's LL-HLS protocol extension. LL-HLS performance relies on selecting the appropriate settings. From latency perspective, our results show that LL-HLS best performs with partial segments with 300 milliseconds or less. Naturally, latency is prone to be less stable in worse network conditions. Startup delay is affected by bitrate and partial segment duration. Network conditions may also significantly affect the startup delay. With very short segment and partial segment durations, the non-media overhead may become a problem. This should be taken into account when selecting media segment and partial segment durations in a large-scale LL-HLS deployment. On the other hand, since in LL-HLS the server refrains from transmitting any bytes belonging to a partial segment until all of its bytes can be transmitted at the full line speed, measuring the available bandwidth is seemingly easier than the systems using CTE (DASH-LL), which subsequently makes rate adaptation easier.

In future work, we plan to further test LL-HLS performance in comparison to other low-latency live streaming protocols. We also plan to further investigate the scalability issues and the impact on the origin servers as well as CDN caches.

Bibliography

[1] "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.

[2] A. C. Begen and Y. Syed, "Are the streaming format wars over?," in *IEEE Int. Conf. Multimedia and Expo Wksp. (ICMEW)*, 2018.

[3] A. Whitepaper, "Ultra-Low-Latency Streaming Using Chunked-Encoded and Chunked-Transferred CMAF." [Online] Available: https://www.akamai.com/us/en/multimedia/documents/white-paper/low-latency-streaming-cmaf-whitepaper.pdf. Online; accessed 5 April 2020.

[4] DASH Industry Forum (DASH-IF), "Low-Latency Modes for DASH." [Online] Available: https://dashif.org/guidelines/. Online; accessed 5 April 2020.

[5] Periscope Code, "Introducing LHLS Media Streaming." [Online] Available: https://medium.com/@periscopecode/introducing-lhls-media-streaming-eb6212948bef. Online; accessed 5 April 2020.

[6] Apple, "Protocol Extension for Low-Latency HLS (Preliminary Specification)." [Online] Available: https://developer.apple.com/documentation/http_live_streaming/protocol_extension_for_low-latency_hls_preliminary_specification. Online; accessed 5 April 2020.

[7] Digital Video Broadcasting (DVB), "ETSI TS 103 285 V1.3.1: MPEG-DASH Profile for Transport of ISO BMFF Based DVB Services over IP Based Networks." [Online] Available: https://www.etsi.org/deliver/etsi_ts/103200_103299/103285/01.03.01_60/ts_103285v010301p.pdf. Online; accessed 5 April 2020.

[8] "ISO/IEC 23009-1:2019 Amendment 1: CMAF support, events processing model and other extensions." [Online] Available: https://www.iso.org/standard/79884.html. Online; accessed 5 April 2020.

[9] "HLS-Announce Mailing List." [Online] Available: https://lists.apple.com/mailman//listinfo/hls-announce. Online; accessed 5 April 2020.

[10] DASH Industry Forum (DASH-IF), "dash.js JavaScript Reference Client." [Online] Available: https://reference.dashif.org/dash.js/. Online; accessed 5 April 2020.

[11] "FFLabs: ffmpeg." [Online] Available: https://gitlab.com/fflabs/ffmpeg. Online; accessed 5 April 2020.

[12] "FFmpeg Formats Documentation." [Online] Available: https://ffmpeg.org/ffmpeg-formats.html#dash-2. Online; accessed 5 April 2020.

[13] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann, "Bandwidth prediction in low-latency chunked streaming," in *ACM NOSSDAV*, 2019.

[14] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann, "Performance analysis of acte: A bandwidth prediction method for low-latency chunked streaming," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 16, June 2020.

[15] Roger N. Pantos, "Low Latency Streaming Media." US16/776808, US16/776824, CN202010100780, CN202010100779. Issued January 30, 2020.

[16] R. Pantos, "HTTP Live Streaming 2nd Edition." [Online] Available: https://datatracker.ietf.org/doc/draft-pantos-hls-rfc8216bis/, Mar. 2020.

[17] Roger N. Pantos, "Preload Hinting for Low Latency HLS." US62/946862. Issued December 11, 2019.

[18] Eryk Vershen and Roger N. Pantos, "Skipping Segments in Playlists." US15/929501. Issued December 11, 2019.

[19] T. Matsuzawa, "Docker test environment for Apple HLS Low Latency Beta Tool." [Online] Available: https://github.com/thmatuza/alhls-tool-docker. Online; accessed 5 April 2020.

[20] Apple, "Using AVFoundation to Play and Persist HTTP Live Streams." [Online] Available: https://developer.apple.com/documentation/avfoundation/media_assets_playback_and_editing/using_avfoundation_to_play_and_persist_http_live_streams. Online; accessed 5 April 2020.

[21] RFC 5905, "Network Time Protocol Version 4: Protocol and Algorithms Specification." [Online] Available: https://tools.ietf.org/html/rfc5905, June 2010.

[22] Akamai, "Test Player for Chunk-Encoded Chunk-Transferred DASH." [Online] Available: http://mediapm.edgesuite.net/will/dash/lowlatency/low-latency-public-example.html. Online; accessed 5 April 2020.

[23] B. Inc., "Video Developer Report." [Online] Available: https://bitmovin.com/bitmovin-2019-video-developer-report-av1-codec-ai-machine-learning-low-latency/, 2019.

[24] R. Pantos, "What's new in Low-Latency HLS." [Online] Available: https://developer.apple.com/videos/play/wwdc2020/10228/. Online; accessed 19 August 2020.

VITA

Kerem Durak received his B.Sc. degree in computer science in 2016. Following his graduation, he worked in telecommunication industry and gained hands-on big data experience. He then started his M.Sc. in Computer Science in Ozyegin University under the supervision of Dr. Ali C. Begen, working on low-latency live video streaming. His interests include Multimedia Networking, Live Streaming and Natural Language Processing.