Real-Time QoE Estimation of DASH-based Mobile Video Applications through Edge Computing

Chang Ge, Ning Wang
5GIC, Institute for Communication Systems
University of Surrey, Guildford, UK
Email: {C.Ge, N.Wang}@surrey.ac.uk

***Abstract*—Video applications using MPEG-DASH (Dynamic Adaptive Streaming over HTTP, such as YouTube and Netflix) have been dominating the Internet traffic in recent years. It is increasingly acknowledged that in order to provide video clients with better Quality-of-Experience (QoE), both content service providers and network operators need to be aware of clients' QoE in the first place. In this paper, we present a novel real-time QoE estimation system through edge computing, which has been implemented and deployed at a real LTE-A network edge. When equipped with such a system, any virtual network function (VNF) deployed in a mobile network will be able to infer all DASH clients' QoE under its coverage in real time, where no feedback from clients are needed. Furthermore, our scheme is able to work robustly in busy network environments involving air interface where packet errors frequently occur. The significance of such a scheme is the availability of accurate and real-time knowledge on user QoE through a very lightweight mechanism at the mobile edge, which can be instantaneously used for various content manipulation or resource adaptation operations in order to assure user QoE in dynamic conditions. Through experiments in a real LTE-A network, we demonstrate that our scheme is able to estimate DASH clients' QoE with very high accuracy with very low CPU and RAM footprint.**

I. Introduction

In recent years, the MPEG-DASH (Dynamic Adaptive Streaming over HTTP) paradigm has been well adopted by content service providers such as YouTube and Netflix to serve videos over the Internet. Its flexibility in terms of letting clients adjust video quality on-the-fly, as well its easy implementation over existing HTTP infrastructure (e.g., Content Delivery Network, or CDN), have contributed to its wide adoption in the video streaming industry.

As videos are increasingly streamed on mobile phones or tablets, it is common that the end-to-end content delivery path involves Radio Access Network (RAN) over the air. Since the RAN condition (e.g., latency, packet error, number of users) fluctuates significantly due to its nature, the user-experienced downlink throughput also fluctuates as a result of TCP's congestion control mechanism [5]. Furthermore, since most DASH clients choose video qualities based on its knowledge on their experienced downlink throughput and/or their buffer level, it is easy for clients to experience either rebuffering (due to overestimating RAN resource availability) or switching video quality frequently. Both cases will lead to suboptimal QoE. Therefore, it is envisaged that if the knowledge of clients' QoE can be obtained in real time during a DASH session, intelligent operations can be performed at the mobile network edge to maintain or improve their QoE. For example, if a DASH client is experiencing poor QoE (e.g., through real-time knowledge on its rebuffering events), or if it is predicted to be experiencing poor QoE (e.g., through real-time knowledge on its video buffer), the mobile network operator (MNO) may adapt its RAN resource allocation so that such a client get more radio resource [12]. Also, the content service provider may adjust its adaptation/caching/prefetching policies or perform TCP congestion window optimization in favor of such a client [3].

Since direct QoE feedback is not implemented in any major DASH client like dash.js[1] or libdash[2], the best option to gain knowledge on clients' QoE is to perform in-network QoE estimation at the network edge. While schemes have been proposed for *offline* reconstruction of QoE [9], it is not a trivial task to perform QoE estimation in real time, especially where RAN is involved. Specifically, real-time QoE estimation means that the start and finish time of each video segment download need to be learned in real time. In a RAN where packet errors such as duplicate ACKs and out-of-order packets frequently take place, the network is not always able to capture all packets. Furthermore, network address translation (NAT) is often used in mobile networks to protect clients' identities through hiding their IP addresses, which makes it difficult to distinguish DASH sessions from each other.

In this paper, we present a novel QoE estimation scheme that is enabled by edge computing. More specifically, our scheme is packaged into a virtual network function (VNF) and is deployed at a Multi-Access Edge Computing (MEC) server, which is in line with the MEC paradigm advocated by ETSI in recent years [4]. Our scheme is able to estimate all 4 major QoE metrics for video applications in real time, including initial playout delay, video buffer, rebuffering events (occurrence and duration) as well as video quality switching statistics. Furthermore, it is able to work robustly where RAN is involved in a content delivery path, which means its functionalities do not rely on any key packet and are not affected by packet errors in the network. It is also able to identify different DASH sessions from each other, even when NAT is used and all DASH clients are exposed with the same IP address.

[1]https://github.com/Dash-Industry-Forum/dash.js
[2]https://github.com/bitmovin/libdash

It is worth noting that when packaged into a VNF, our scheme can be deployed/operated by different stakeholders. On one hand, it can be operated by the MNO within its own infrastructure to estimate its subscribers' video QoE. On the other hand, it can be deployed within the MNO network and is operated by the content service provider. This is feasible since the content service provider can rent computing and storage resources within an MNO infrastructure and deploy its service capability and intelligence (e.g., QoE estimation) there [2].

To the best of our knowledge, our key contributions in this paper are as follows:

- Our proposed MEC-enabled QoE estimation scheme is the first that works robustly in a RAN environment in real time (where packet errors frequently occur) with high accuracy and low CPU and RAM footprint.
- Our scheme is the first of its kind to be packaged into a VNF and deployed in a real LTE-A network infrastructure, which serves as a use case for the MEC paradigm. Furthermore, the detailed descriptions on its implementation provide important practical guidelines in realizing QoE awareness at mobile network edge.

II. Background Review

When a video is disseminated under the MPEG-DASH standard [7], it is encoded/compressed into different representations (i.e., video qualities) with different bitrates. Each representation is then divided into multiple segments with identical length in seconds. The metadata about a video's representations and their segments (e.g., URLs) are stored in a manifest file named media presentation description, or MPD file. When a client streams a video, its MPD manifest is first requested so that the client can understand the video's available representations and their structures. Afterwards, the client makes decision on which representation to stream on a per-segment basis. There are 4 main metrics that govern the QoE in a DASH session [8]:

- **Video buffer**: the available video buffer at a client.
- **Initial playout delay**: the time duration between when a client clicks on "play" and when the video starts playing at the client.
- **Rebuffering**: the number and duration of rebuffering (i.e., playback freezing) events during a session.
- **Video quality switching statistics**: the quality (bitrate) of the segments consumed in a DASH session, and the number of events where video quality switches between segments.

In order to obtain the knowledge of a DASH client's QoE, the existing techniques in the literature can be classified into 3 categories: 1) direct client feedback; 2) in-network QoE estimation and 3) mapping QoS parameters (e.g., bandwidth, latency etc.) to QoE in Mean Opinion Score (MOS) [8]. In this paper, we focus on the second category, which consists of *real-time* in-network QoE estimation techniques [10] and *offline* techniques [11] [9].

We specifically review the technique in [9], because even though it works in offline timescale, it is the only technique that is able to estimate all 4 QoE metrics above, which is done through analyzing packet sniffing traces from within the network. Specifically, every time a client downloads a video segment, this technique utilizes 3 key packets during the download to estimate the client's QoE. These key packets contain the segment's HTTP GET request, HTTP 200 OK response and the last packet containing the requested segment data respectively. When used in an offline analysis, due to the complete context in the packet trace, these packets can be easily identified and used to accurately reconstruct the start and finish of each segment download, which are further used to calculate the client's buffer conditions and whether any rebuffering event has occurred. However, in order to achieve the same objective in real time, especially when RAN is involved in the content delivery path, there are multiple challenges that need to be tackled.

First, in order to estimate a client's buffer level on-the-fly, the QoE estimation function needs to capture the start and finish time of each video segment download *in real time*. This means the QoE estimation algorithm needs to be very lightweight so that it not only analyzes packets fast enough, but also has low CPU and RAM footprint so that it can operate in a stable manner over long term.

Second, in a RAN where out-of-order packets, duplicate ACKs and TCP retransmissions take place frequently, the QoE estimation algorithm needs to be able to handle these events while processing captured packets. Furthermore, in a very busy network, packets may get dropped by kernel and are not 100% captured. Therefore, the algorithm needs to work robustly and should not rely on any key packet to perform QoE estimation.

Third, in an MNO infrastructure, NAT is often deployed to hide the client's IP addresses and protect their identities. In this case, the QoE estimation function needs to be able to distinguish clients from each other, even when all clients share the same IP address in packet traces.

III. System Overview

The system architecture is illustrated in Figure 1. Specifically, we consider the scenario where a DASH client streams a video from a content source (owned by content provider) via an LTE network infrastructure (owned by MNO), and a MEC server is deployed within the MNO infrastructure. Note that we use the LTE architecture as an example only - the proposed technique can be applied to any access network technology. As shown in the figure, when our QoE estimation VNF is deployed at a MEC server, it consists of 4 functional blocks:

HTTP proxy is responsible for handling incoming DASH requests by e.g., resolving them to an appropriate video source. It provides input to the QoE estimation algorithm in terms of incoming requests' URL and timestamp. Our scheme works for both forward (transparent) and reverse (non-transparent) proxies when the DASH session is unencrypted. However, if the session is encrypted, our scheme works only on reverse proxies since the requests' URLs need to be decrypted at an SSL/TLS connection's termination point. A reverse proxy

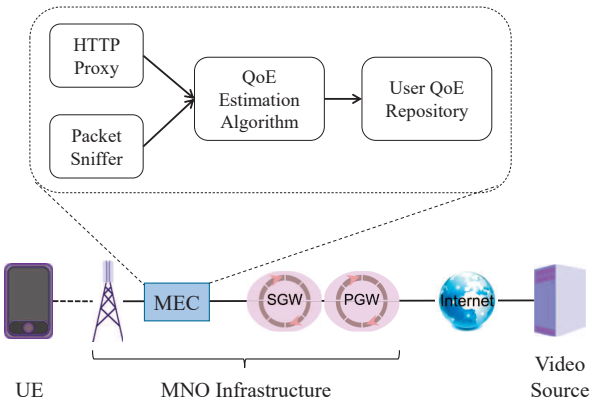

Fig. 1. System Architecture

can be operated by either the MNO or the content provider, depending on their business models.

Packet sniffer captures packets on transport layer and identifies all DASH-related ones. It then pre-processes each packet's information into an efficient one-line format, which includes timestamp, TCP port number, SEQ/ACK number etc. (more details are shown in Section IV). The information is then passed to the QoE estimation algorithm as inputs. Note that our algorithm needs packets' information at up to transport layer only - it does not need any information from the HTTP layer. This means it works for both unencrypted (http) and encrypted (https) sessions.

We use *tcpdump* to perform packet sniffing in our QoE estimation scheme, since it is crucial that the packet sniffer captures packets as fast as possible in a lightweight manner, so that it does not fall behind the actual packet flow. Although *tshark/wireshark* is able to automatically detect TCP retransmissions and out-of-order packets etc., the fact that it stores historic packet states means it is slower than stateless packet sniffers like *tcpdump*. Also, this causes its RAM usage to keep increasing indefinitely. In contrast, when using *tcpdump*, all it needs to do is to identify and capture DASH-related packets and offload the task of identifying packet errors to the QoE estimation algorithm. Furthermore, its kernel buffer-based mechanism means its RAM usage is capped.

QoE estimation algorithm processes the inputs from the HTTP proxy and the packet sniffer, and estimates all 4 QoE metrics mentioned earlier of all DASH clients covered by the MEC server (more details on the algorithm are described in Section IV). It also deposits the estimated QoE knowledge into a user QoE repository.

User QoE repository stores all DASH clients' current and historic QoE knowledge, which can be used by MNO and/or content service provider to perform QoE-driven optimizations as discussed in Section I.

Also, the physical location of the QoE estimation VNF is flexible within the MNO infrastructure - it can be deployed anywhere between a base station and the P-GW, as long as there is the capability to host such a function.

IV. QoE Estimation Algorithm

A. DASH Session Identification

When the HTTP proxy receives a video segment request, the QoE estimation algorithm needs to first identify which DASH session this request belongs to. Similarly, when the packet sniffer captures a packet, the algorithm also needs to 1) check if this is a DASH-relevant packet; and 2) if yes, which DASH session does it belong to. In order to perform such a task, the QoE estimation algorithm maintains a list of all DASH clients, where each client contains a list of DASH video sessions (VS), where a VS instance is created every time an MPD manifest request is sent from the client. Furthermore, each VS contains a list of video segment download sessions (SDS), where an SDS instance is created every time a segment request is sent from the client.

When the HTTP proxy receives a segment request, its belonging client is identified based on its source IP address. It is then assigned to a VS instance within that client based on the following criteria. Note that any VS that has already finished playback are not considered - a VS is set to finished when its last video segment has been delivered to the client.

- The VS whose last request's segment ID and source TCP port are the closest to the incoming request's segment ID and source TCP port.
- The VS whose video's name matches the requested video's name. This does not apply to those URLs which do not contain video's name.

When the packet sniffer captures a packet, its belonging DASH client is first identified based on its source /destination IP address. Its VS instance is then identified based on its source/destination TCP port number in the same way as criteria #1 above. The other criteria is not applicable since the packet sniffer does not use the URL information. In order to identify the packet's belonging SDS instance, we use the timestamp when the HTTP proxy receives an HTTP GET request of a segment, named "**REQ**", to mark the beginning of an SDS in the packet trace. All packets that belong to the same DASH client and VS instances are captured after an SDS's REQ timestamp are assigned to that SDS. Such a strategy is based on the fact that the *packet* containing the GET request may not always be captured, may arrive out-of-order, or may be duplicated. By adopting a key timestamp-based strategy, these packets errors are effectively ruled out.

It is worth mentioning that the DASH client/VS/SDS identification mechanism above works when a client has multiple ongoing DASH sessions. Therefore, it works if the client is a computer streaming multiple videos at the same time (e.g., through multiple web browser tabs). Furthermore, it works if the actual clients are behind NAT, which means they all share the same IP address (i.e., the NAT gateway's). This significantly improves its practicality in real-world applications.

B. Video Quality Estimation

In order to estimate a DASH client's experienced video quality, the algorithm needs to know the exact URL that the

client requested for each segment, which is then mapped into specific representations as follows. At the beginning of each DASH session, the video's MPD manifest is requested by the client. When the MPD file is sent from the video source to the client, the HTTP proxy keeps a copy of the file and parses the URL of each representation's each segment and generates a look-up table. Later on, every time the client requests a segment, the HTTP proxy is able to map the URL contained in the HTTP GET request into a specific segment ID and representation. Hence, the video quality and switching events are captured with 100% accuracy on a per-segment basis.

We use the HTTP proxy (instead of using the packet sniffer) to obtain the knowledge on requested URLs. This is because if the DASH session is encrypted, due to how modern encryption schemes like TLS v1.2 work [6], the packet sniffer will not be able to decrypt the URL contained in the GET request on-the-fly. On the other hand, the HTTP proxy is able to decrypt the request because it is an SSL/TLS termination point as explained earlier.

C. DASH Client Video Buffer Estimation

In order to estimate a DASH client's video buffer in real time, the algorithm needs to detect the following 2 key timestamps within each SDS. They need to be identified as soon as possible as the packet flow passes the packet sniffer. Note that both initial playout delay and rebuffering events are estimated with real-time knowledge of a client's video buffer.

The first key timestamp, **LAST_SENT**, marks the point when the last packet containing data of the requested segment is sent towards the client. When an SDS's REQ timestamp is identified, a counter is started to record how many bytes have been sent towards the client within this SDS. A **SENT** packet is identified to contain a requested segment's data sent towards the client if the following criteria are met:

- It has been identified to belong to the same DASH client, VS and SDS instances.
- Its destination IP address and TCP port number match the GET request's source address and port number.
- Its TCP PUSH flag has been set.

Every time a SENT packet is identified, its TCP SEQ number is used to update the above counter. When the counter reaches the size of the requested video segment (which the HTTP reverse proxy knows by design), the latest SENT packet's timestamp is identified as the SDS's LAST_SENT. Until when the next REQ is identified, all subsequent SENT packets are ignored since they are considered to be duplicate or out-of-order. Note that we do not simply use the packet's length to update the counter, since that would include out-of-order packets and especially TCP retransmissions, which would produce inaccurate results.

The second key timestamp, **LAST_ACK**, marks the point when the HTTP proxy receives the last ACK sent from the client, hence acknowledging it has received the entire requested video segment. An **ACK** packet is identified to contain a client's ACK response while downloading a video segment if the following criteria are met:

- It has been identified to belong to the same DASH client, VS and SDS instances.
- It has the same source IP address and TCP port number as the GET request's source IP address and TCP port number.

Since an ACK packet is always captured after its SENT counterpart, the identification of an SDS's LAST_ACK starts after when its LAST_SENT has been identified. Specifically, an SDS's LAST_ACK is identified to be the timestamp of the ACK packet whose TCP ACK number is equal to the LAST_SENT packet's SEQ number.

We now explain how these 2 key timestamps are used to estimate a DASH client's video buffer in Algorithm 1 below.

Algorithm 1: Video Buffer Estimation Algorithm

Input: *sdsList*: a list containing all SDS whose both key timestamps have been identified
currTime: current timestamp
Output: *buffer*: the VS's current video buffer in s
rebufferDur: the VS's rebuffering duration in s
rebufferCount: the VS's rebuffering occurences

```
1:  begin
2:  buffer, rebufferDur, rebufferCount ← 0
3:  prevDS ← null
4:  for each SDS in the list
5:    currSDS ← currently SDS in iteration
6:    currSDSFinishTime ←
            (currSDS's LAST_SENT + LAST_ACK)/2
7:    if currSDS's segment ID > 1
8:      buffer -= (currSDSFinishTime - prevSDSFinishTime)
9:      if buffer < 0
10:       rebufferDur += buffer × -1
11:       rebufferCount += 1
12:       buffer ← 0
13:     end if
14:   end if
15:   if this is the last SDS in sdsList
16:     buffer -= (currTime - currSDSFinishTime)
17:   end if
18:   buffer += currSDS's length
19:   prevSDS ← currSDS
20: end for
21: if buffer < 0
22:   rebufferDur += buffer × -1
23:   rebufferCount += 1
24:   buffer ← 0
25: end
```

Algorithm 1 is executed periodically on each VS of each DASH client to estimate its current buffer by iterating through its *sdsList*. An SDS is added to *sdsList* as soon as both of its key timestamps are identified. Specifically, at line 6, it estimates the actual time when the client has received all packets of the SDS, which is the mid-point between LAST_SENT and LAST_ACK of the SDS (assuming similar uplink / downlink latencies). At lines 7-14, it calculates the buffer value on

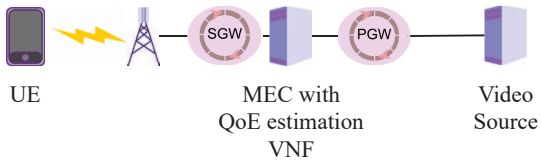

Fig. 2. Experiment setup

a rolling basis. It also updates the VS's rebuffering metrics between any 2 SDSs, because if the buffer result turns out to be negative, it means rebuffering has occurred. At lines 15-17, it deducts the gap between the current time and the last SDS's finish time from the buffer. At line 18, it adds the last finished SDS's length to the buffer. Finally, at lines 21-25, it updates the VS's rebuffering statistics if the final buffer result is negative after iterating through all SDSs in the list.

Both the DASH client/VS/SDS identification mechanism in Section IV-A and Algorithm 1 runs in linear time and space complexity, which is determined by the number of *active* VSs present. In the next section, we demonstrate that the proposed QoE estimation algorithm can achieve very high accuracy while consuming very low CPU and RAM resources.

V. Performance Evaluation

A. Experiment Setup

In order to evaluate the proposed QoE estimation scheme's performance, we conduct experiments in a real LTE-A network testbed (hosted by the 5G Innovation Centre at University of Surrey, UK). We use a PC running Ubuntu 16.04, Google Chrome browser v62 and dash.js v2.6.3 as the DASH client, which is connected to an LTE Band 41 small cell via a Huawei Nexus 6P mobile phone through USB tethering. The client streams a video with 246.08s length that is first compressed into 1 single representation with H.264 at 15Mbps, and then divided into DASH segments of 2s length each.

The QoE estimation scheme is implemented and packaged into a VNF and deployed onto a MEC server, which is deployed next to the Packet Gateway (P-GW) in the LTE-A core network. More specifically, the HTTP reverse proxy is implemented using Jetty[3]; the packet sniffer uses *tcpdump* to capture packets; the QoE estimation algorithm itself is implemented with Java; and the QoE repository is implemented using MySQL.

B. Evaluation of Buffer Estimation Accuracy

In order to evaluate the accuracy of the proposed QoE estimation algorithm under various network conditions, we conducted experiments under 3 scenarios with different backhaul latency (20ms, 150ms and 300ms) and packet error rate (0%, 0.05% and 0.1%) between the MEC server and the video source. The RAN conditions are the same among the 3 scenarios. In these experiments, we extract the JavaScript console logs from the Chrome browser to reconstruct the real buffer conditions every 0.5s as a benchmark. These logs record the start and finish time of each segment download with 1ms precision, which accurately reflects the actual buffer condition at the DASH client. Our QoE estimation algorithm was also configured to output its inferred buffer values to the QoE repository every 0.5s, which are exported afterwards for comparison. Note that we do not evaluate the accuracy of estimated video quality, since it is always 100% accurate via URL matching in the MPD manifest.

[3]eclipse.org/jetty

TABLE I
Comparison between real and estimated QoE statistics

		Initial Delay	Playback Duration	Rebuffering Count / Time
(a)	Real	0.663s	246.786s	0 / 0s
	Estimated	0.512s	246.52s	0 / 0s
(b)	Real	2.699s	248.696s	0 / 0s
	Estimated	2.557s	248.704s	0 / 0s
(c)	Real	2.99s	252.965s	7 / 3.875s
	Estimated	2.84s	252.79s	8 / 3.85s

The comparison between the real and estimated buffer length are shown in Figure 3. It is observed that the blue (estimated) and red (real) lines are very close to each other under all 3 scenarios. We further show the statistics of real and inferred QoE metrics in Table I. It is shown that the proposed algorithm achieved very high accuracy when estimating initial playout delay, playback duration and rebuffering duration. Specifically, the estimation error on the initial delay, playback duration and rebuffering duration are up to 0.151s, 0.266s and 0.025s respectively, which are up to 0.1% of the video length.

Note that in scenario (c), the algorithm inferred one more rebuffering event than the real situation between around 210s and 230s. This took place because the QoE estimation algorithm runs every 0.5s in the experiments. Therefore, if a segment download finished right after the algorithm, it will not be included in the buffer calculation process for up to another 0.5s. Hence, if the buffer is already lower than 0.5s and the algorithm just "missed" a finished segment download, it would create a false rebuffering event which will correct itself in the next iteration. Nevertheless, the very-high accuracy of the proposed algorithm on the other QoE metrics has been validated through these experiments.

C. Evaluation of CPU and RAM usage

In order to evaluate the proposed QoE estimation scheme's CPU and RAM usage, we follow the same setup in Figure 2 and used 10 Huawei Nexus 6P phones to continuously stream 10 different videos at 15Mbps each, and these 10 concurrent DASH sessions are maintained over a 24-hour period. Note that while the aggregated 150Mbps traffic may not seem high, a MEC server is typically expected to cover a small area and hence a small number of users that streams 4K video [1].

The results on CPU and RAM usages are plotted in Figure 4. It is shown that the CPU usage is maintained at a low

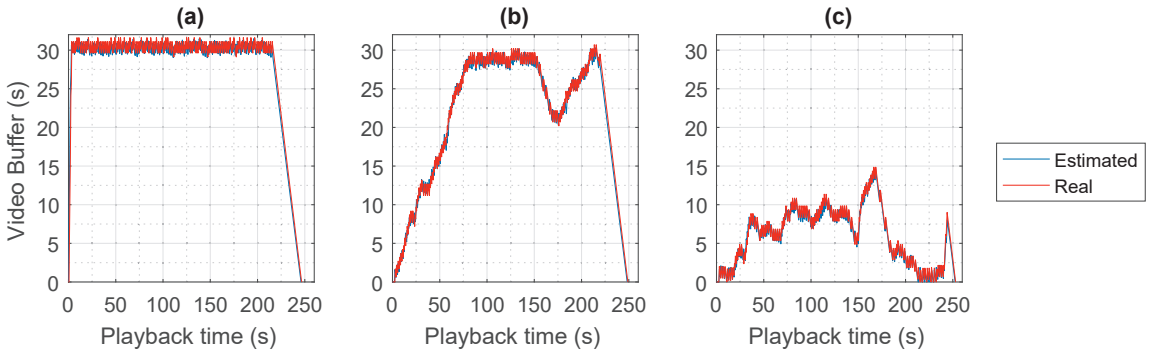

Fig. 3. Comparison between estimated and real buffer length under (a) ideal; (b) fluctuating and (c) poor network conditions

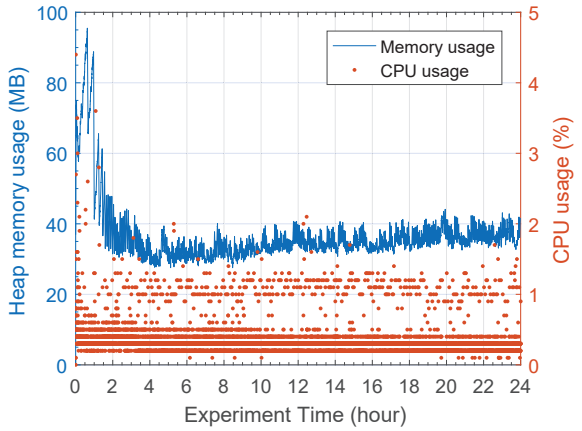

Fig. 4. CPU and RAM usage over 24 hours

level with an average of 0.32%. Regarding the RAM usage, it is higher in the beginning due to program initialization that occupies more memory, which are eventually released through Garbage Collection. After the initial period, it reduces to a lower level and stays steady with an average of 36.87MB. These results validate our early analysis that the proposed algorithm's time and space complexity are linear to the number of *active* DASH sessions. They also validate that the proposed scheme is capable of operating in a lightweight and stable manner over long term.

VI. Conclusion

In this paper, we have proposed a novel scheme that estimates DASH video applications' QoE in real time through edge computing. Unlike existing schemes in the literature, its algorithm does not rely on capturing any key packet in a DASH session. Furthermore, it is capable of ruling out packet errors (e.g., out-of-order or duplicate packets) by inspecting TCP and IP headers only, which means it works robustly in a RAN environment where packet error frequently occur. It is also capable of distinguishing video sessions behind NAT.

Through experiments in a real LTE-A network testbed, we have demonstrated that our proposed scheme is capable of estimating initial playout delay and rebuffering duration while achieving an accuracy within up to 0.266s. Furthermore, we have validated that its CPU and RAM usage are both very low and steady over a 24-hour experiment. As the first QoE estimation VNF that has been packaged and deployed in a real MEC server, we are confident that our work not only serves as a use case for the MEC paradigm, but also provides practical guidelines towards realizing QoE awareness in mobile networks.

VII. Acknowledgments

This work is funded by EPSRC KCN project (EP/L026120 /1). The authors would also like to acknowledge the support of the University of Surrey's 5G Innovation Centre (5GIC) (http://www.surrey.ac.uk/5gic) members for this work.

References

[1] Mobile Edge Computing - a key technology towards 5G.
[2] Akamai Technologies. Akamai introduces predictive video over cellular capabilities (press release). Available at https://www.akamai.com/uk/en/about/news/press/2015-press/akamai-introduces-predictive-video-over-cellular-capabilities.jsp.
[3] A. E. Essaili, D. Schroeder, E. Steinbach, D. Staehle, and M. Shehada. QoE-based traffic and resource management for adaptive HTTP video delivery in LTE. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(6):988–1001, June 2015.
[4] ETSI. Mobile Edge Computing: A key technology towards 5G (Whitepaper). Available at http://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf.
[5] C. Ge, N. Wang, G. Foster, and M. Wilson. Toward QoE-assured 4K video-on-demand delivery through mobile edge virtualization with adaptive prefetching. *IEEE Transactions on Multimedia*, 19(10):2222–2237, Oct 2017.
[6] IETF. RFC5246 - the transport layer security (TLS) protocol version 1.2, 2008.
[7] ISO. ISO/IEC 23009-1:2014 dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats.
[8] P. Juluri, V. Tamarapalli, and D. Medhi. Measurement of quality of experience of video-on-demand services: A survey. *IEEE Communications Surveys & Tutorials*, 18(1):401–418, Firstquarter 2016.
[9] R. Huysegems et al. Session reconstruction for HTTP adaptive streaming: Laying the foundation for network-based QoE monitoring. In *Proc. IWQoS*. IEEE, June 2012.
[10] S. Latré et al. On-line estimation of the QoE of progressive download services in multimedia access networks. In *Proc. ICOMP*, pages 181–187, 2008.
[11] R. Schatz, T. Hosfeld, and P. Casas. Passive YouTube QoE monitoring for ISPs. In *Proc. IMIS*, pages 358–364. IEEE, July 2012.
[12] S. Thakolsri, W. Kellerer, and E. Steinbach. QoE-based cross-layer optimization of wireless video with unperceivable temporal video quality fluctuation. In *Proc. ICC*. IEEE, June 2011.