A Wrapper for Automatic Measurements with YouTube's Native Android App

Michael Seufert*, Bernd Zeidler*, Florian Wamser*, Theodoros Karagkioules†,
Dimitrios Tsilimantos†, Frank Loh*, Phuoc Tran-Gia*, Stefan Valentin†
*University of Würzburg, Institute of Computer Science, Würzburg, Germany
{seufert | bernd.zeidler | florian.wamser | frank.loh | trangia}@informatik.uni-wuerzburg.de
†Huawei Technologies, France Research Center, Paris, France
{theodoros.karagkioules | dimitrios.tsilimantos | stefan.valentin}@huawei.com

***Abstract*—YouTube is one of the most popular and demanding services in the Internet today. Thereby, a large portion of this traffic is generated by YouTube's mobile app. While past studies have shown how to monitor browser-based streaming on desktop PCs (e.g., YoMo) or mobile devices (e.g., YoMoApp), streaming in the native app has not been monitored yet. This paper presents an automated framework for monitoring the streaming in YouTube's native app for Android. The concept is based on a wrapper application and the Android Debug Bridge (adb), and can be also extended to automatic measurements with other apps. For YouTube, it allows to collect application-layer streaming data, such as current playtime, buffered playtime, video encoding, and quality switches. These data can be complemented with network measurements on the mobile access link to obtain a holistic view on mobile YouTube streaming on Android devices. In addition to describing the software design and testbed setup, this paper discusses an experimental measurement. This study analyzes the streaming in the native YouTube app and compares it to the streaming from the mobile YouTube website via YoMoApp.**

I. Introduction

The mobile YouTube app brings one of today's most popular and volume-dominant services on the Internet to mobile devices. Out of the hundreds of millions of hours watched and billions of views generated on YouTube every day, more than half of the service requests come from mobile devices[1] and their perceived service quality might suffer from fluctuating network conditions in constrained and stressed mobile networks. Still, the goal of video streaming services and (mobile) network operators is to increase their revenue by achieving a high Quality of Experience (QoE), which is a concept used to describe the subjectively perceived quality of end users with an Internet service [1], [2]. While video streaming services, such as YouTube, can influence the video content preparation (e.g., coding format, video bit rate) and the adaptation of the video quality (i.e., HTTP Adaptive Video Streaming (HAS)), network operators employ QoE-aware traffic management within the constrained networks to meet SLAs for data transmissions, to reduce their costs by sophisticated utilization of the network resources, and to reach a high end user satisfaction.

To evaluate the current situation in the network, which is the basis of all traffic management, network operators rely on QoE monitoring to assess the satisfaction of end users and use the monitored information for traffic management decisions. Typically, they distribute hardware middleboxes or virtual network functions in their networks, which investigate the traffic, extract relevant features, and estimate the QoE. However, the recent trend towards end-to-end encryption reduces the visibility of monitoring approaches in the network. Instead, solutions are required, which estimate the QoE from features that were extracted from the encrypted packet data, e.g., [3]. To build accurate QoE estimators for streaming apps, such as YouTube, a deep understanding of the underlying network- and application-layer characteristics of the app is required.

Therefore, this paper presents an automated framework for monitoring the streaming in the native Android YouTube app in a testbed. The concept relies on a wrapper app based on the Android Testing Support Library and the Android Debug Bridge (adb), making it customizable to other Android apps. For YouTube, it allows to collect all application-layer streaming data like current and buffered playtime, stalling, video coding format, and quality switches. These data are complemented with network measurements on the mobile access link to obtain a holistic view on mobile YouTube streaming on Android devices, making it the first approach that can provide such detailed insights. The presented wrapper app is available on GitHub[2]. In an exemplary study, the streaming of the YouTube app is analyzed in three different network conditions with two different videos to show the applicability of the presented QoE measurement concept. Moreover, the streaming with the YouTube app is compared to the streaming from the mobile YouTube website via YoMoApp [4], [5]. A larger data set measured from two different vantage points with different devices and video content is available[3], but will not be discussed in this work due to space limitations. In future works, data sets generated with our measurement framework will be further evaluated to build accurate QoE estimators for YouTube streaming, which work on encrypted network traffic.

The remainder of this paper is structured as follows. Section II introduces related works on network and QoE measurements for video streaming. Section III presents the novel monitoring approach. Section IV describes the testbed and measurement setup, and the results are presented in Section V. Finally, Section VI concludes this paper.

[1] https://www.youtube.com/yt/about/press/ – All webpages referenced in this paper were accessed on May 8, 2018.

[2] https://github.com/lsinfo3/yomo-wrapperapp

[3] http://qoecube.informatik.uni-wuerzburg.de

II. Related Work

Several tools have been proposed in literature to measure the performance of Internet applications, such as [6], [7]. With special attention to the measurement of video streaming apps, [8]–[10] monitored streaming with a browser plug-in, and were able to measure several QoE metrics, such as initial delay, stalling, and adaptation. In [11], the authors present a system for on-line monitoring of YouTube QoE in cellular networks using in-network measurements only. [12] implemented an evaluation tool for YouTube QoE in Android mobile devices. However, this application did not take adaptive video streaming into account. In [13], a mobile application was presented to actively measure and analyze mobile app QoE on network and application layer. In [14], the authors introduce a tool to measure and analyze mobile app QoE, based on active measurements on network and application layers.

YoMoApp (YouTube Performance Monitoring Application) [4], [5], is a unique Android application, which passively, non-intrusively monitors application- and network-level KPIs of YouTube adaptive video streaming on mobile devices, and can collect subjective QoE feedback of users. The monitored KPIs and collected ratings can be used to analyze the QoE of mobile YouTube video sessions. Therefore, an Android WebView browser element is embedded to display the YouTube mobile website, on which YouTube HTML5 video streaming is possible. Monitoring functions are added, which query the HTML5 ⟨*video*⟩ object to obtain player state/events, buffer, and video resolution. Additionally, network statistics (e.g., RAT, transmitted bytes) and context parameters can be monitored in the native Android part of YoMoApp, and optional subjective QoE feedback can be obtained after a streaming session.

III. QoE Monitoring of the Native Android YouTube App

QoE monitoring of YouTube adaptive video streaming at a mobile device is difficult because internal data of the client app cannot be easily accessed. Custom made apps, such as YoMoApp [4], [5], worked around and streamed from the mobile YouTube website with a browser, where it is possible to measure all relevant QoE parameters. Still, researchers are unsure if there is a difference of the streaming from the mobile YouTube website via YoMoApp and the native YouTube app due to multiple versions and frequent updates of the YouTube streaming environment (i.e., different streaming for different devices/operating systems/app versions). As the original app is prevailing, it is highly desirable to perform QoE measurements directly with the native app on regular mobile devices. Therefore, a measurement approach for the Android YouTube app is presented using an Android debug interface and a wrapper app. The approach leverages that many relevant streaming information is displayed in the YouTube app. This includes the current video playback time and video duration in the progress bar, and a special feature of the YouTube app called *"stats for nerds"*, which allows the user to display advanced streaming information, e.g., buffer size, video format, or network speed estimation (cf. Figure 1 and Table I).

To extract these data from the app, first of all, a direct communication to the device has to be established. The *Android Debug Brigde* (*adb*)[4] is a software interface for Android, which can be used to connect Android devices to a PC via USB, and thus, does not interfere with the Internet connection of the device via Wi-Fi or cellular network. With adb, the PC can act as a controller and can send commands to the Android device in order to trigger events. Most importantly, the *input* command simulates an input event on the Android device like a key press, a touch event, or a swipe motion. The basic syntax for a touch event, as an example, is *adb shell input tap x y* . With *x* and *y*, a position on the screen is defined using the exact pixel coordinates. Using this command, the controlling PC can emulate every user interaction with the Android device. In the context of the Android YouTube App, this approach is capable to interact with the app and the streaming just like a user can do, e.g., search for a specific video, start or pause the video playback, or change the video resolution. This allows to control the measurement in the YouTube app. During the streaming, the "Copy to clipboard" button can be tapped periodically, in order to copy the "stats for nerds" values to the system clipboard from where it can be stored to a file with the help of an additional app. Later the file can be transferred to the controlling PC via adb. To access the user interface, on which the video progress bar is displayed, Android provides a tool for dumping the structure and contents of the active window in

Fig. 1: Screenshot from YouTube app during measurement with *"stats for nerds"* and timestamp above the seek bar.

TABLE I: YouTube "stats for nerds" debug information.

Key	Presumable meaning	Example
csdk	Android SDK version	"csdk":"25"
c	Platform type	"c":"android"
cbrand	Phone brand	"cbrand":"Google"
cbrver	App version	"cbrver":"12.34.55"
cplayer	Player type	"cplayer":"ANDROID_EXOPLAYER"
cplatform	Player platform	"cplatform":"mobile"
cmodel	Phone model	"cmodel":"Pixel XL"
cver	App version	"cver":"12.34.55"
cbr	App package name	"cbr":"com.google.android.youtube"
cosver	OS version	"cosver":"7.1.1"
cos	OS name	"cos":"Android"
videoid	Video id	"videoid":"N2sCbtodGMI"
cpn	⟨meaning unknown⟩	"cpn":"IahGa5wh6X9rUhD-"
fmt	Video format id (itag)	"fmt":"244"
afmt	Audio format id (itag)	"afmt":"140"
bh	Buffered playtime	"bh":34900
bwe	Bandwidth estimate	"bwe":4740424
conn	Number of connections	"conn":3
bat	Battery charge	"bat":"0.330:1"
df	Ratio of dropped frames	"df":"0/166"
timestamp	Timestamp	"timestamp":"2017-09-22T06:36:08.005Z"
glrenderingmode	Player rendering mode	"glrenderingmode":"RECTANGULAR_2D"
innertube.build	Build information	"innertube.build.changelist":"169439144", "innertube.build.experiments.source_version": "169565248","innertube.build.label": "youtube_20170920_0_RC5", "innertube.build.timestamp":"1505939237", "innertube.build.variants.checksum": "28e1cc6b7674123a48c07976d04819f9"
e	⟨meaning unknown⟩ (aka fexp)	"e":"23700451,23700497,23700647, 23701914,23701958,23702322,23702700, 23702708,23702739,23702753,23702894, 23703156,23703299,23703890,23704018, 23704045,23704248,23704263,23704600, 9405973,9415293,9422596,9431754, 9435797,9444108,9444635,9449243, 9450141,9456940,9463607,9463830, 9464088,9467503,9470292,9476026, 9476327,9477614,9478727,9480475, 9481734,9482942,9484344,9484377, 9485000,9488038,9488474,9489124"
logged_in	User log-in state	"logged_in":"1"

[4]https://developer.android.com/studio/command-line/adb

form of an XML file, called *uiautomator*[5]. Using the command *adb exec-out uiautomator dump /dev/tty* allows the controlling PC to retrieve the activity content of the currently active application, which contains the video progress bar, as well as the current playback time and video duration.

As extensive communication with the controlling PC via adb, e.g., frequently triggering touch events or dumping the contents of the active window, can cause delay and reliability issues, most of the controlling algorithm should be moved to the mobile device. Only basic commands should be exchanged during the measurement, such as starting the monitoring and interacting with the application, while the actual QoE monitoring should be performed autonomously on the device. Therefore, the chosen approach relies on the Android *Testing Support Library*[6], which provides an extensive testing framework for automated user interface testing of Android applications. While the original purpose of the framework is testing in-development applications, it can be applied to any third-party application as a controlling *"wrapper app"*. Thereby, the framework sends an additional app to the Android device, which can "wrap" around any other app on the device and interact with it. Note that it is not necessary to change the controlled application, which makes this approach also applicable for researching other Android applications.

Using the wrapper app also provides the ability to perform actions directly on UI elements inside of an application instead of relying on position controlled touch events. Additionally, information about the UI can be collected directly on the device. Elements can be accessed using simple filters utilizing descriptions, IDs, or content text. This does not only decouple the control mechanism from the screen resolution and exact position of UI elements, but also allows the controlling PC to check if an action has been executed correctly. Concurrent operations can be coordinated by testing the availability of the required buttons, avoiding any colliding calls to UI elements. Moreover, the wrapper app operates autonomously on the phone, and communication with the controlling PC is only needed for higher level instructions.

In case of measuring the native Android YouTube app, the wrapper app subsequently launches the YouTube app, opens the settings menu, disables autoplay, enables "stats for nerds", sets the starting quality (if required), starts the video playback, and opens the "stats for nerds" UI. During the video playback, two sets of information are passed to the controller. First, the video progress is obtained by querying the description of the "SeekBar" object in the YouTube UI. It is transmitted directly to the controller over a TCP connection, running through an adb forward. On the controlling PC, the video progress information is logged to a file. The progress information can be sampled roughly in intervals of 500 ms. Second, the "stats for nerds" data is obtained. Therefore, an additional Android app is used, which dumps the contents of the clipboard into a file on the phone. Every second, the wrapper app performs a click action on the "stats for nerds" UI "Copy to clipboard" button, and invokes the installed clipboard app over a broadcast intent. The resulting "stats for nerds" logfile is copied to the PC after the measurement.

[5]https://developer.android.com/training/testing/ui-automator

[6]https://developer.android.com/topic/libraries/testing-support-library/

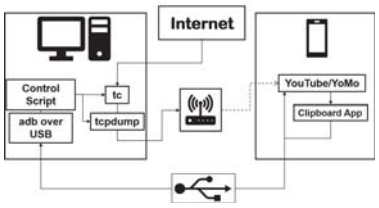


Fig. 2: Schematic representation of the testbed

IV. Study Description

The goal of the study was to prove the applicability of the proposed measurement approach. Therefore, a testbed was set up and the streaming of two different videos with the native YouTube app was measured in three different network scenarios. The measured streaming data of the YouTube app are evaluated both in terms of network usage and resulting Quality of Experience. Moreover, the streaming of the native YouTube app is compared to the streaming of YoMoApp [4], [5], an Android app designed for monitoring YouTube video streaming, which uses the mobile YouTube website in a WebView browser for streaming. The results can give insights if both approaches can reliably measure the same YouTube service, or if and how the streaming differs for different services (native app, mobile website) offered by YouTube. Therefore, both apps are used to stream the same videos in the same testbed under the same network conditions.

The testbed employed in the measurements is installed at the University of Würzburg and consists of a Linux PC, currently running *Ubuntu 14.04.4 LTS*, a Google Pixel running *Android 7.1.1*, and a Wi-Fi dongle, and is shown in Figure 2. During the measurement, the phone is connected to the PC over a USB connection, as well as to a Wi-Fi access point set up on the PC via the Wi-Fi dongle. The USB connection provides power and a means to control the phone using adb. The Wi-Fi connection connects the phone to the Internet via the Internet link of the PC. Thus, the Wi-Fi connection is utilized by the phone to stream the video, but can be controlled and monitored by the PC. Therefore, two additional programs are utilized in the measurement process on the PC. These are the Linux internal *tc* with *netem* for shaping the traffic on the Internet link of the PC, and *tcpdump* for monitoring the network traffic of the smartphone on the wireless link.

To allow for a varied set of measurements, both the used videos and the traffic shaping plans can be configured easily through a file with special syntax. This file specifies the YouTube video ID and defines selected quality level, bandwidth limit, latency, and packet loss for each measurement. These values can be applied for the whole video, or can be altered during the streaming using a measurement itinerary. Another persistent configuration file is used for setting the measurement conditions at each testbed site. It contains technical information that may vary between sites, such as network interface name, phone ID, measurement configuration, and communication ports. The main controlling script reads these configuration files and adjusts the testbed accordingly.

During the experiments performed in this study, three network scenarios have been used, i.e., *Unlimited*, *Varying*, and *Limited*. The *Unlimited* scenario did not apply any traffic shaping. As the testbed is connected via university network to

TABLE II: Network scenario *Varying*.

Measurement Time	Bandwidth	Delay	Packet Loss
start - 120 s	-	-	-
120 s - 160 s	5000 kbps	11 ms	1.00%
160 s - 200 s	4000 kbps	13 ms	1.25%
200 s - 240 s	3000 kbps	15 ms	1.50%
240 s - 280 s	2000 kbps	20 ms	1.75%
280 s - 320 s	1500 kbps	25 ms	2.00%
320 s - 360 s	2000 kbps	20 ms	1.75%
360 s - 400 s	3000 kbps	15 ms	1.50%
400 s - end	4000 kbps	13 ms	1.25%

the optical fiber network of the German National Research and Education Network (DFN), high throughputs of up to around 20 Mbps could be achieved. The *Limited* scenario used *tc* with hierarchical token bucket for limiting the bandwidth to 500 kbps during the whole measurement. The *Varying* scenario featured changing network conditions in terms of bandwidth limitations, delay, and packet loss. It follows a reference network profile[7] defined by the DASH Industry Forum with some modifications with respect to the timing intervals, and is presented in detail in Table II. Two videos were used with different characteristics, which were not monetized, i.e., no advertisement clips were shown with the video. Video 1 (YouTube ID *D8YQn7o_AyA*, duration 17:54 min) was the German TV broadcast of the penalty kicks of the UEFA EURO 2016 quarter finals between Germany and Italy, which was available in all quality levels up to 2160p, and Video 2 (YouTube ID *Y-rmzh0PI3c*, duration 12:10 min) was the animated movie "Cosmos Laundromat - First Cycle" from the Blender Foundation, which was available in quality levels up to 1080p. Note that for all measurements, the quality level was not specified during the streaming, i.e., the automatic quality level selection feature of YouTube was used.

V. Results

A measurement produces three output files: a tcpdump of the network traffic, a "stats for nerds" log, and a playtime log. The tcpdump log is a textual representation of a packet capture trace (pcap) of all TCP/QUIC traffic of the YouTube app including a Unix timestamp, information about IP address and port of the source and destination, packet length, protocol version, sequence numbers, and flags for each packet. It also includes the DNS requests and responses, such that IP addresses can be mapped to domain names. The recognized domain names can be used to identify network ranges of YouTube video servers (e.g., googlevideo.com) or ads servers (e.g., doubleclick.net). Thus, information about the content in each flow are given, even if the flow itself is encrypted.

The "stats for nerds" log also contains a Unix timestamp and several comma-separated key-value-pairs of data in quotation marks. They are listed in Table I with their presumed meaning and an example. It can be seen that information about the phone and the player are logged, as well as playback information, such as the video and audio itag (fmt, afmt), the buffered playtime in milliseconds (bh), and the number of played out frames, which is the denominator in the ratio of dropped frames (df). The itag of the video format can be looked up to obtain information about the video codec, the resolution, and the frame rate. Thereby, the number of played out frames could be converted into the current video playtime. The video playtime can also be directly obtained from the displayed progress bar of the video player. These data are periodically stored in the progress log, which simply lists the current Unix timestamp in milliseconds, the current video playtime, and the video duration.

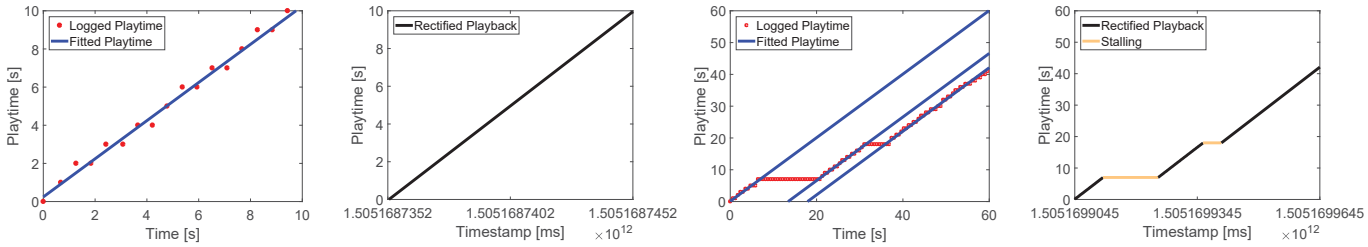
[7] http://dashif.org/wp-content/uploads/2016/06/DASH-AVC-264-Test-Vectors-v1.0.pdf

A. Postprocessing of the Measured Data

Postprocessing is an important step after the measurements. It validates the data and extracts relevant parameters from raw data by preparing and processing the logfiles.

1) Playtime Fitting with Linear Function: The playtime log has to be postprocessed as it only contains discrete playtime values (integer seconds). These values correspond to the measured data at a given time value. The discrete values of the logfile represent the playtime as a step function, which does not reflect the continuous playback. Therefore, a linear fitting is applied to the raw data to obtain a continuous playtime function. The linear fitting is required to have a slope of 1, as the video playtime should advance one second during one second of real time (unit rate constraint). Figure 3 shows a measured playtime function and its rectified version. In Figure 3a, the red dots show the integer raw data and the blue line presents the corresponding linear fit. The linear fitting gives the estimated values for the playtime, which can be used for further analyses, and can be used to obtain the rectified playtime function, which is depicted in Figure 3b and returns the current video playtime at any given timestamp. In case of playback interruptions, i.e., stalling, the methodology has to be only slightly modified. A stalling period is detected as a time period with constant playtime. Due to the step nature of the raw playtime function and the periodic polling with an interval of 500 ms, a stalling threshold of 1 s is introduced. If the derivative of the playtime function is 0 for a period larger than that threshold, the period is considered a stalling period. For each period of playback, the linear fitting with slope 1 can be applied, which is depicted in Figure 3c. After piecewise reconstruction, Figure 3d shows the corresponding rectified playback function, which allows to obtain exact timestamps for begin and end of stallings (yellow).

2) Labeling: In addition to the timestamps of playback start, stalling start, and stalling end, the timestamps of quality changes can be obtained directly from the "stats for nerds" log file by observing changes of the format itags (fmt/afmt). To label network-related events, the tcpdump network log is postprocessed. First, all DNS requests and responses are extracted, as well as the start and end times, durations, and sizes of all included flows, which are described by a four-tuple of source and destination IP addresses and ports. The DNS requests/responses are evaluated to identify video server IP addresses or network ranges based on known video domain names (e.g., googlevideo.com). The timestamp of the first packet to a video server is labeled as the initial request, which allows to obtain the initial delay of the streaming session, i.e., the difference between the timestamps of the playback start and the initial request. Moreover, the buffering phases, i.e., the periods in which video data are downloaded, can be identified from inspecting the video flows. Thereby, video flows are defined as flows, whose source IP address belongs to a video server. To set the end of a buffering



(a) Fitting of playtime log. (b) Rectified playback. (c) Detection of stalling, piecewise fitting of playtime log. (d) Rectified playback and stalling.

Fig. 3: Fitting of logged playtime and rectified playback.

phase, a threshold of 1 s is used, i.e., the timestamp of the buffering end event is set in case no video packets arrive in the following second. To obtain the buffered playtime, the cumulative download function, i.e., the total amount of downloaded video data, is used. The downloaded playtime at any time can be computed from the cumulative download function after externally downloading the video in the used format itags with *youtube-dl* and inspecting segment sizes and contained playtime with *ffprobe*. The buffered playtime can then be computed as downloaded playtime minus current playtime. Note that also the "stats for nerds" log contains the buffered playtime (bh) and can be used for validation, but the network method provides a finer granularity.

B. Comparison of Streaming in YouTube App and YoMoApp

The streaming of the YouTube app was analyzed in the testbed for three different network scenarios, i.e., *Unlimited*, *Varying*, and *Limited*, and two selected videos, i.e., Video 1 (*D8YQn7o_AyA*) and Video 2 (*Y-rmzh0PI3c*). It is compared to the streaming from the YouTube mobile website via YoMoApp. Each combination of network scenario, video, and application was measured ten times in the same testbed in January 2018. Thereby, for each combination, the repetitions showed a very similar streaming behavior, implying that the testbed is able to produce representative results. Figure 4 shows plots of exemplary streaming sessions of Video 1 for each network condition. In each plot, the x-axis shows the time from the start of the measurement. The left y-axis and black line depict the download throughput per second in Mbps, the brown dashed horizontal line indicates the average throughput, and the right y-axis and orange curve show the cumulative download volume in MB. The six plots are arranged in three columns, which represent the three network conditions, and two rows. The streaming with YouTube app is depicted on top, and the corresponding streaming with YoMoApp on bottom.

In the *Unlimited* scenario of Video 1, the YouTube app shows an initial phase, in which it constantly downloads data with throughputs of around 18 Mbps (black). After around 30 s and almost until the end of the video, there is a regular pattern of short download bursts of around 1 s in intervals of roughly 10 s for both videos. Thus, the download rate is significantly reduced, which results in a well visible "bend" of the cumulative download volume (orange) between the two phases. Thereby, the first phase can be considered a "filling phase", in which the video buffer shall be filled until a certain level of buffered playtime. A best effort download is used, and thus, the download rate in this phase is only limited by the network. In the second "steady state" phase, the download rate is reduced to maintain a constant level of buffered playtime, such that new video data are only downloaded when the buffered playtime drops below a threshold. For both measured videos, this threshold was 120 s. This means that one burst contains the data for a playtime of 10 s, and thereby, the download rate approximates the video bit rate. This can be followed from the unit rate constraint of the video playback (one second of playtime is played out in one second of real time), i.e., to maintain a constant buffer, the fill rate (download rate) of the buffer is equal to the depletion rate of the buffer (video bit rate). The reason to reduce the download rate might be to not stress servers with best effort downloads, and the level of the constant buffer might be selected in order to trade-off the risk of network degradations with the amount of unnecessarily downloaded video data in case of video abortion. The total downloaded data is 367.3 MB, the average throughput is 3.011 Mbps, and the download ratio is 36.10%, i.e., downloads were active in 36.10% of the time on average. Video 2 shows a very similar behavior. Only due to a lower video bit rate, the duration of the filling phase is shorter and the total downloaded data (108.1 MB), average throughput (1.302 Mbps), and download ratio (23.67%) are smaller.

The beginning of the *Varying* scenario is similar to *Unlimited*, and thus, the streaming of Video 1 also starts with a filling phase and reaches the steady state phase. When the bandwidth limitations start, the bursts become longer, as more time is needed to download the required video data of one burst. After 240 s (cf. Table II), the download rate is obviously lower than the video bit rate, which can be seen in the orange curve, and the streaming returns to filling phase, in which it is constantly downloading with a rate limited by the network conditions. During that time the buffered playtime decreases to roughly 72 s. When the bandwidth increases, the desired buffer level of 120 s is again reached, and the streaming switches to steady state phase. The total amount of downloaded data and average throughput were similar to *Unlimited*, only the download ratio is higher than *Unlimited* (79.64%). Video 2 shows a similar behavior, but due to the lower video bit rate, the second filling phase is later and shorter. Again only the download ratio is higher compared to *Unlimited* (43.59%).

In the *Limited* condition, the streaming of Video 1 shows a similar behavior to *Unlimited* with a target buffered playtime of 120 s, however, a much longer filling phase can be observed (ca. 180 s) due to the bandwidth limitation. During this phase, two changes of itag (quality adaptation) occur. The quality is eventually switched down to a much lower video bit rate,

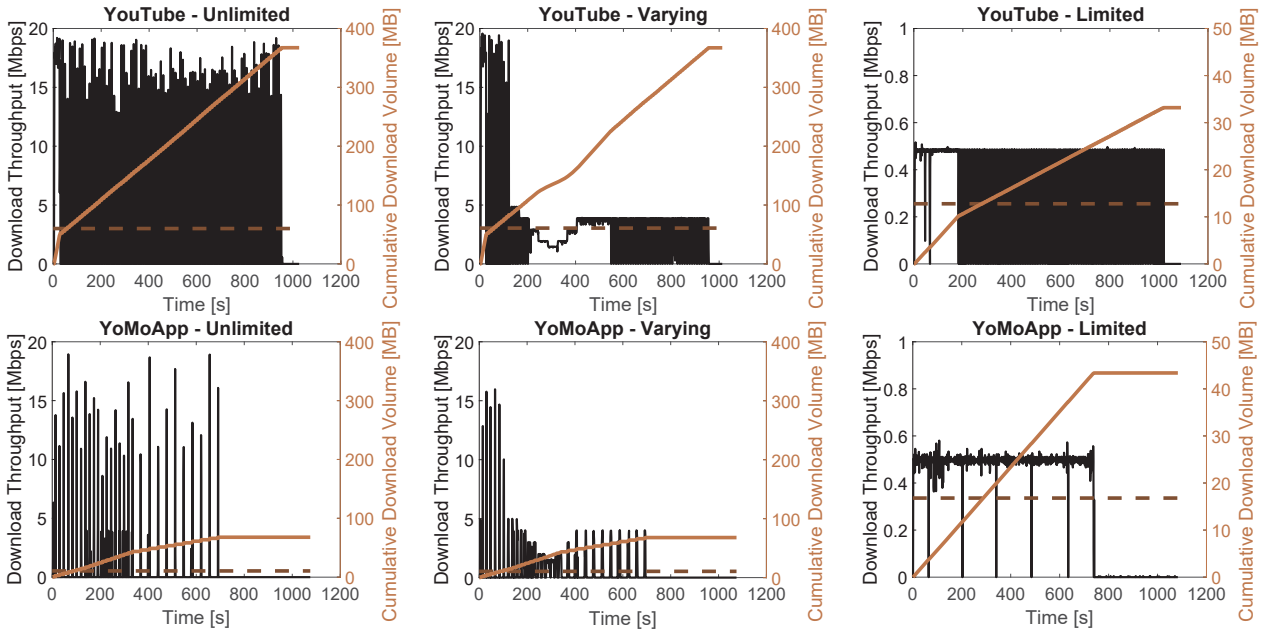


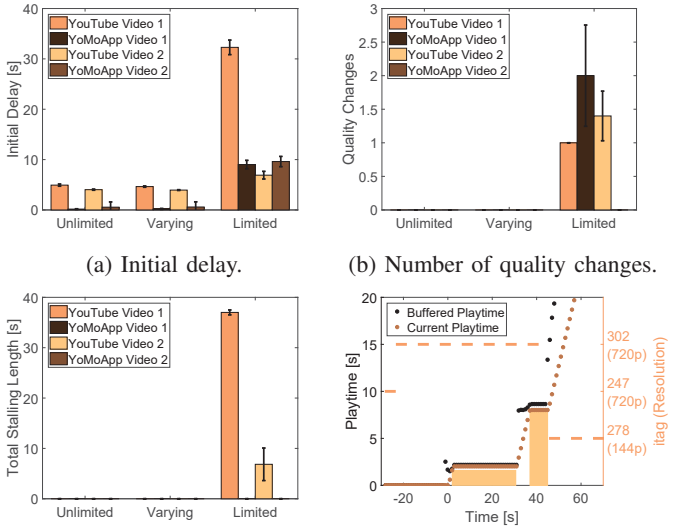
Fig. 4: Throughput (black), average throughput (dashed brown), and cumulative download volume (orange) of representative streaming sessions of Video 1 (*D8YQn7o_AyA*) with YouTube app (top) and YoMoApp (bottom) in all investigated scenarios.

which results in a much lower downloaded volume, but also in a reduced visual quality (see QoE results below). The total amount of downloaded data is 33.18 MB, the average download throughput is 254 Kbps, and the download ratio is 64.20%. Video 2 again shows a very similar behavior with a shorter filling phase. The total downloaded data is 18.20 MB, the average download throughput is 217 Kbps, and the download ratio is 56.04%.

Comparing the streaming of the mobile YouTube website with YoMoApp, some differences are visible. In the *Unlimited* scenario of Video 1, YoMoApp shows only regular and large download bursts of around 2 MB every 18 s. After 335 s the interval doubles to around 36 s and a slight bend can be recognized in the orange curve. In this steady state phase, YoMoApp maintains a buffered playtime threshold of 310 s, which is much higher than in the YouTube app. The total downloaded volume is 67.85 MB, which is much smaller than in YouTube app, and indicates that a lower video bit rate and lower visual quality was streamed. The average throughput is 530 Kbps, and the download ratio is 14.94%, both also smaller than for YouTube app. For Video 2, the total downloaded volume is 35.11 MB, the average throughput is 402 Kbps, and the download ratio is 14.35%. However, the streaming phases could not be identified. Instead, the streaming showed the same large bursts interleaved with small bursts of around 500 KB. This might suggest that also the large bursts of 2 MB that could be seen for both videos are actually composed of four small bursts. However, there might also be a more complex relation between the video content, video bit rate, segment length, and burst size. In the *Varying* scenario of both videos, the bandwidth limitation is also visible and causes longer and more frequent bursts. Eventually, the streaming reaches the steady state phase with longer intervals between large bursts. While the download volume and average throughput are similar, the download ratios are higher reaching 25.40% for Video 1, and 22.02% for Video 2. In the *Limited* scenario, the streaming of YoMoApp shows an almost constant download with average download ratio 69.97% for Video 1 and 81.40% for Video 2. The download rate is limited by the network condition only and is a best effort download with an average throughput of 333 Kbps for Video 1 and 395 Kbps for Video 2. Interestingly, a higher amount of data is downloaded than with YouTube app (43.03 MB for Video 1 and 34.87 MB for Video 2).

The results confirm that both streaming services provided by YouTube are not very similar. The streaming in the YouTube app clearly shows two phases. There is a filling phase with enduring best effort download (download rate limited by network) until a certain threshold of buffered playtime is reached, and a steady state phase, in which a constant buffer is maintained by a regular on/off-downloading pattern with small bursts of data (download rate limited by video bit rate). In contrast, streaming from the mobile YouTube website with YoMoApp rather showed a best effort download in larger intervals with larger bursts, although phases with different download rates exist as well. Most prominently, the total amount of downloaded data is much lower in YoMoApp in *Unlimited* and *Varying* scenario, which indicates that the YouTube app offers a higher visual quality of the streamed video. In the *Limited* scenario, YoMoApp streamed with a constantly active download and downloaded only a slightly lower amount of data than in the other scenarios. As the YouTube app significantly reduced the amount of downloaded data below the amount of YoMoApp, YoMoApp might provide a higher visual quality in this scenario. In the following, the QoE-related parameters are investigated in more detail.

Figure 5 shows several QoE-related parameters of the streaming sessions for each of three network scenarios. All bars represent the mean values with 95% confidence intervals over the ten repetitions for each combination on top. Thereby, orange bars show results of the native YouTube app, and brown bars results of YoMoApp. The darker and lighter colors represent Video 1 and Video 2, respectively. Figure 5a shows the initial delay of the streaming session, i.e., the time from the



(a) Initial delay.

(b) Number of quality changes.

(c) Total stalling length.

(d) Video 1 buffer and playback in *Limited* scenario (YouTube).

Fig. 5: Evaluation of Quality of Experience results.

video request to the begin of the playback. In the *Unlimited* scenario, YouTube app shows a mean initial delay of 4.93 s for Video 1 and 4.03 s for Video 2. Similar values can be seen for the *Varying* scenario (4.64 s for Video 1 and 3.95 s for Video 2), which has the same network conditions as *Unlimited* in the first 120 s, i.e., no bandwidth limitation. In case of the *Limited* condition, the mean initial delays increase up to 32.29 s for Video 1 and 6.91 s for Video 2. Taking a closer look at the "stats for nerds" log, it could be seen that the initially requested itag for Video 1 was 302 in *Unlimited* and *Varying* scenarios, which had a total video size of 344.3 MB, and it was itag 247 in the *Limited* scenario, which had a total size of 203.2 MB. Note that both itags 302 and 247 use VP9 coding format in a webm container and have a resolution of 720p[8] but the average encoding bit rates obviously differ (2.689 Mbps for itag 302, 1.587 Mbps for itag 247). Video 2 was initially downloaded in all scenarios in itag 247 with a total size of only 95.02 MB and average bit rate of 1.092 Mbps, and this smaller average video bit rate might be responsible for the shorter initial delays.

Figure 5b presents the mean number of quality changes, and Figure 5c shows the mean total stalling length, respectively. It can be seen that no quality change or stalling occurred in *Unlimited* and *Varying* scenarios. Thereby, Video 1 was requested all the time in itag 302, and Video 2 was requested all the time in itag 247. However, quality changes and stallings occurred in the *Limited* scenario. Video 1 always showed one quality change and two stalling events. Stalling shows that the selected video bit rate (itag) cannot be supported by the limited network conditions, and consequently, YouTube requests a lower quality. Figure 5d shows the first seconds of an exemplary streaming of Video 1 in *Limited* condition. The current playtime (brown) and the available playtime (black) are plotted on the left y-axis, while the orange dashed line shows the displayed itag and resolution on the right y-axis. The x-axis depicts the time from the playback start in seconds, so negative times indicates the initial delay. In this network condition, YouTube initially requested itag 247 and changed to request itag 302 after around 5 s, still during the initial delay. Note that this quality change is not counted in Figure 5b as it cannot be perceived by the users. Interestingly, this quality change constitutes an increase of video bit rate. The reason is that the bandwidth is initially estimated as 18 Mbps (bwe, cf. "stats for nerds" log), probably based on the previous measurements with that device. It takes more than 40 s until the estimation is corrected to 500 Kbps. The reason could be that the estimation is only updated after a download burst is finished.

When YouTube started the playback, only 2.5 s of playtime, which might be the initial delay threshold, could be played out in itag 302. The downloaded data was not sufficient, and caused the first stalling event, which is illustrated by the yellow area. When another 5 s of playtime, which might be the stalling threshold, were downloaded, the playback resumes at 32 s until 37 s. At this point, YouTube realized that the limited network condition did not support the high bit rate of the segment, so YouTube switched to itag 278, which has a total size of 12.78 MB, a much lower average bit rate of 100 Kbps, and the lowest video resolution of 144p. The reason for the late switch might be that YouTube app uses a fixed number of connections, which might initially all download parts of the high quality video. After the bandwidth estimation was corrected, YouTube app possibly still had to wait until a download burst was finished and its connection was available to download the new itag. After the download of another 5 s of playtime, playback continued at 46 s and the low bit rate of this itag could be supported by the network for the remainder of the session. This streaming behavior was consistent over all ten runs.

Also for Video 2, the initial itag is 247 and the final itag is 278 with total size of 5.561 MB and average bit rate of 64 Kbps. However, either one or two quality changes occur. Four times, there was a first quality change after 20 s either to itag 244 (480p, three times) or 243 (360p, once), and a second quality change to itag 278 after another 17 s (for all itag 244) or 14 s (for itag 243). Six times, there was only one quality change to itag 278 either after 20 s (two times) or 42 s (four times) of playtime. The exact timing of the quality switches in both videos can be explained by the segment borders of YouTube's HAS system, i.e., segments contain a fixed amount of playtime to enable switches at certain points in the video. Only in the case of switching to the lowest quality after 20 s, stalling could be avoided. For the other eight measurements of Video 2, YouTube did not manage to switch to the lowest quality before stalling occurred (one stalling event). These results suggest that stalling events in YouTube app occur when the bandwidth estimation is wrong and a too high video quality was selected. Thereby, the bandwidth estimation might also consider historical streaming data. Moreover, stalling events are closely related also to quality changes, as they might constitute the "initial delay" of the download of a new itag.

Compared to the YouTube app, the mean initial delays of YoMoApp are much shorter with 0.20 s for Video 1 and 0.59 s for Video 2 in *Unlimited* and *Varying* scenario. Considering the total download volume, this can be explained by the much lower bit rate requested in YoMoApp. The YoMoApp logs show that, in *Unlimited* and *Varying* scenario, Video 1 is always streamed in 360p, and Video 2 in 240p, but the itags cannot be accessed in YoMoApp. In the *Limited* scenario the initial delays are 9.01 s for Video 1 and 9.61 s for Video 2 due

[8] https://godoc.org/github.com/rylio/ytdl

to the bandwidth limitation to 500 Kbps. However, YoMoApp also starts the streaming of Video 1 with 360p. After one to four quality switches to 240p or 144p, the final resolution of 240p is reached. In contrast, Video 2 is streamed in 240p all the time. Thus, both videos can maintain a higher resolution than the resolution to which YouTube app converged (144p), while avoiding stalling in all scenarios.

In general, it can be seen that the streaming of YouTube app and the mobile YouTube website with YoMoApp deliver different streaming QoE. In case of sufficient bandwidth, YouTube app streams a much higher video bit rate, which results in a higher time on higher quality layer, and thus, an improved QoE [2], [15]. When the bandwidth is limited, stallings can occur with the YouTube app, to some extend caused by wrong bandwidth estimation and new initial delays when the quality level (video bit rate) is changed and a new itag is requested. Moreover, a very conservative behavior was observed as YouTube eventually switched to the lowest quality layer for the remainder of the session. YoMoApp, in contrast, better coped with the bandwidth limitation. It adapted the quality better and could maintain a higher quality level without any stalling, which results in a higher QoE [2].

VI. Conclusion

This paper presented a novel QoE measurement concept for the native Android YouTube app, which is based on a wrapper app and the Android Debug Bridge. In a testbed with a controlling PC, user interactions within the YouTube app can be emulated and displayed information can be logged. As the concept is based on the Android Testing Support Library, it can be easily extended to research also other Android apps.

After postprocessing of the logged raw data and the captured network trace, all playback- and network-related parameters of the YouTube streaming could be monitored. These include the timestamps of initial request, playback start, starts and ends of stalling, starts and end of buffering phases, and quality changes. Moreover, current playback time, buffered playback time, video and audio encoding formats, and other parameters of YouTube's "stats for nerds" feature could be monitored. Thus, the presented concept allows to obtain valuable insights into the streaming and the resulting QoE of YouTube users on Android devices, which can be used, e.g., to develop accurate QoE estimators for encrypted video traffic.

In an exemplary measurement study, the streaming in the YouTube app was compared to the streaming from the mobile YouTube website with YoMoApp. The presented analyses show that the novel measurement approach allows to gain excellent insights in the streaming with the native Android app both on application and network level. It allows to drill down on numbers and performance measures to reach a detail level that is unprecedented by previous measurement approaches. It could be seen that both streaming services offered by YouTube for mobile devices (Android app and mobile website) show significantly different streaming behaviors. The YouTube app clearly showed an initial filling phase and a steady state phase with regular small bursts to maintain a certain level of buffered playtime, while YoMoApp rather showed a best effort download in larger intervals with larger bursts. Also the QoE results for both streaming services were different, as the YouTube app provided a higher QoE in case of sufficient bandwidth. However, YoMoApp could cope better with limited bandwidth, because YouTube app showed a wrong bandwidth estimation, which was based on historical streaming data.

To measure mobile YouTube streaming on network and application layer, the wrapper app approach should be preferred, as it can automatically measure the popular native Android YouTube app and provides very detailed insights into the streaming both on network and application layer. However, the wrapper app approach can only be used in a testbed because it requires adb/USB connection to a controlling PC. In contrast, YoMoApp can be used by real users in the field to measure the YouTube streaming from the mobile website. Although the streaming might be different to the native YouTube app, YoMoApp additionally allows to collect subjective ratings, which is beneficial for the QoE research of video streaming.

References

[1] P. Le Callet, S. Möller, and A. Perkis (eds), "Qualinet White Paper on Definitions of Quality of Experience," COST Action Qualinet, Tech. Rep., 2013.

[2] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, "A Survey on Quality of Experience of HTTP Adaptive Streaming," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, 2015.

[3] D. Tsilimantos, T. Karagkioules, A. Nogales-Gómez, and S. Valentin, "Traffic Profiling for Mobile Video Streaming," in *International Conference on Communications (ICC)*, 2017.

[4] F. Wamser, M. Seufert, P. Casas, R. Irmer, P. Tran-Gia, and R. Schatz, "YoMoApp: a Tool for Analyzing QoE of YouTube HTTP Adaptive Streaming in Mobile Networks," in *European Conference on Networks and Communications (EuCNC)*, 2015.

[5] M. Seufert, N. Wehner, F. Wamser, P. Casas, A. D'Alconzo, and P. Tran-Gia, "Unsupervised QoE Field Study for Mobile YouTube Video Streaming with YoMoApp," in *Conference on Quality of Multimedia Experience (QoMEX)*, 2017.

[6] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the Edge Network," in *Internet Measurement Conference (IMC)*, 2010.

[7] A. Nikravesh, H. Yao, S. Xu, D. Choffnes, and Z. M. Mao, "Mobilyzer: An Open Platform for Controllable Mobile Network Measurements," in *Conf. on Mobile Systems, Applications and Services (MobiSys)*, 2015.

[8] B. Staehle, M. Hirth, R. Pries, F. Wamser, and D. Staehle, "YoMo: A YouTube Application Comfort Monitoring Tool," in *Wrkshp. of Quality of Experience for Multimedia Content Sharing (QoEMCS)*, 2010.

[9] R. K. P. Mok, E. W. W. Chan, and R. K. C. Chang, "Measuring the Quality of Experience of HTTP Video Streaming," in *Symposium on Integrated Network Management (IM)*, 2011.

[10] H. Nam, K.-H. Kim, D. Calin, and H. Schulzrinne, "YouSlow: A Performance Analysis Tool for Adaptive Bitrate Video Streaming," *ACM SIGCOMM Computer Communication Review*, vol. 44, 2014.

[11] P. Casas, M. Seufert, and R. Schatz, "YOUQMON: A System for Online Monitoring of YouTube QoE in Operational 3G Networks," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, 2013.

[12] G. Gómez, L. Hortigüela, Q. Pérez, J. Lorca, R. García, and M. C. Aguayo-Torres, "YouTube QoE Evaluation Tool for Android Wireless Terminals," *EURASIP Journal on Wireless Communications and Networking*, vol. 164, 2014.

[13] Q. A. Chen, H. Luo, S. Rosen, Z. M. Mao, K. Iyer, J. Hui, K. Sontineni, and K. Lau, "QoE Doctor: Diagnosing Mobile App QoE with Automated UI Control and Cross-layer Analysis," in *Internet Measurement Conference (IMC)*, 2014.

[14] A. Schwind, M. Seufert, Ö. Alay, P. Casas, P. Tran-Gia, and F. Wamser, "Concept and Implementation of Video QoE Measurements in a Mobile Broadband Testbed," in *Network Traffic Measurement and Analysis Conference (TMA)*, 2017.

[15] M. Seufert, T. Hoßfeld, and C. Sieber, "Impact of Intermediate Layer on Quality of Experience of HTTP Adaptive Streaming," in *Conference on Network and Service Management (CNSM)*, 2015.