Livelyzer: Analyzing the First-Mile Ingest Performance of Live Video Streaming

Xiao Zhu
University of Michigan

Subhabrata Sen
AT&T Labs – Research

Z. Morley Mao
University of Michigan

ABSTRACT

Over-the-top (OTT) live video traffic has grown significantly, fueled by fundamental shifts in how users consume video content (e.g., increased cord-cutting) and by improvements in camera technologies, computing power, and wireless resources. A key determining factor for the end-to-end live streaming QoE is the design of the first-mile upstream ingest path that captures and transmits the live content in real-time, from the broadcaster to the remote video server. This path often involves either a Wi-Fi or cellular component, and is likely to be bandwidth-constrained with time-varying capacity, making the task of high-quality video delivery challenging. Today, there is little understanding of the state of the art in the design of this critical path, with existing research focused mainly on the downstream distribution path, from the video server to end viewers.

To shed more light on the first-mile ingest aspect of live streaming, we propose Livelyzer, a generalized active measurement and black-box testing framework for analyzing the performance of this component in popular live streaming software and services under controlled settings. We use Livelyzer to characterize the ingest behavior and performance of several live streaming platforms, identify design deficiencies that lead to poor performance, and propose best practice design recommendations to improve the same.

CCS CONCEPTS

• **Information systems** → **Multimedia streaming**; • **Networks** → **Network measurement**.

KEYWORDS

live streaming, video ingest, performance, QoE, measurement

ACM Reference Format:
Xiao Zhu, Subhabrata Sen, and Z. Morley Mao. 2021. Livelyzer: Analyzing the First-Mile Ingest Performance of Live Video Streaming. In *12th ACM Multimedia Systems Conference (MMSys '21) (MMSys 21), September 28-October 1, 2021, Istanbul, Turkey*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458305.3463375

1 INTRODUCTION

Live video streaming traffic has grown significantly, fueled by improvements in camera technologies, computing power, and wireless resources. The rise of services such as Facebook and Youtube creates global platforms to disseminate user-generated content. According to a recent industry report [24], live video will account for more than 15% of the Internet video traffic by 2022.

An end-to-end (E2E) live streaming pipeline consists of the ingest and distribution paths shown in Figure 1. On the upstream ingest path, the video is captured in real time by a camera, then fed into a *Broadcasting App* that compresses the video and transmits it to a remote *Video Server* owned by some streaming service over a network connection, typically cellular or Wi-Fi. On receiving the ingest stream, the video server transcodes it into a number of different ABR tracks (referred to as ABR track ladder), each corresponding to a different encoding quality level and bitrate. Each track consists of several video segments (usually 2–10 seconds each). Viewers watching the live stream request a mixture of segments from the video server using adaptive bitrate (ABR) streaming [46] over the downstream distribution path.

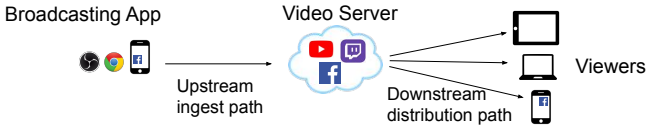


Figure 1: Live video streaming end-to-end workflow.

Existing studies have focused largely on the last-mile distribution path from the video server to the viewers. There has been little exploration of the first-mile ingest path from the broadcasting app to the video server. However, this first mile is critical to the E2E performance of the pipeline. The quality of the video delivered on this first mile to the video server imposes an upper limit on the quality of the ABR tracks created from it, and therefore on the quality of experience (QoE) of the viewers of the live stream. In addition to delivering a good quality video stream to the video server, the first mile also needs to provide the content with low latency. Any latency on the first mile impacts the overall E2E latency for the end viewers (see §2.1). Improving the ingest performance would therefore benefit the QoE of all the downstream viewers. However, achieving this goal is also challenging due to the usually more dynamic and limited wireless uplink resources (*e.g.,* cellular uplinks) and the complexity of the ingest path.

In this paper, we examine the all-important first-mile ingest path in commercial live streaming platforms to understand their performances and designs. Such insights can assist developers in identifying deficiencies and creating designs with improved performance and network providers to better understand and manage the associated traffic [21, 59, 60]. Our goal is to analyze a wide range of commercial live video broadcasting apps and streaming services

MMSys 21, September 28-October 1, 2021, Istanbul, Turkey

ACM ISBN 978-1-4503-8434-6/21/09…$15.00
https://doi.org/10.1145/3458305.3463375

from an objective third-party point of view, in a controlled, repeatable, and fine-grained manner (§2.2). This task is made challenging by the complex E2E pipeline, the proprietary closed-source software components, and the wide diversity of designs across different live streaming systems. The live nature of the content introduces further challenges in conducting measurements (see §2.4).

In view of these challenges, we develop a generalized black-box measurement methodology and tool, Livelyzer, for analyzing the performance of the upstream ingest path for commercial live streaming systems. Livelyzer enables third parties to conduct active measurements to profile the performance under various network conditions in a repeatable and controlled manner, thereby gaining insights into the corresponding design. The design of Livelyzer and its capabilities are detailed in §3.

We use Livelyzer to study a wide range of broadcasting apps such as third-party, browser-based, and mobile-based broadcasting apps streaming to commercial services including Facebook, Youtube, and Twitch, using different video contents and network conditions. In total, we study seven (broadcasting app, streaming service) combinations. Our key findings are:

• Different broadcasting apps have very different encoding rate control designs/configurations. Many of them use Constant Bit Rate (CBR) encoding, leading to inefficient use of bits on the upstream ingest path (§4.1).

• Different broadcasting apps behave very differently when the network conditions change. Our evaluations show that while all the broadcasting apps we study exhibit adaptation to changing uplink network conditions, they differ widely in the specific adaptation behavior and resulting ingest performance. Further, our results suggest that the existing adaptation strategies have limitations and sometimes lead to poor performance. For example, the Open Broadcaster Software (OBS), by default, drops frames in an inefficient way to cope with network condition degradation, leading to poor video quality (§5.1.1). Although OBS recently introduced a "dynamic bitrate mode" for encoding rate adjustment, we find that the scheme can largely under-utilize the newly available network resources due to how it adapts the encoding bitrate when the network bandwidth increases (§5.1.2). Browser-based (§5.2) and mobile-based (§5.3) broadcasting apps also exhibit performance issues.

• We leverage Livelyzer to conduct a what-if analysis of the rate adaptation logic and codec usage, in order to understand the impact of different rate adaptation schemes and video codecs on the live video ingest performance (§6). We show how even a relatively straightforward adaptation strategy inspired by the findings of Livelyzer can help improve the performance (§6.1). We further assess using a more efficient H.265 codec (widely supported in major desktop and mobile platforms) instead of the current commonly used H.264, for live video encoding in broadcasting apps and demonstrate its benefits under different network uplink conditions (§6.2).

• The video server design also has implications for QoE. For example, we find that different services choose different segment durations, making the ingest delay variable across different broadcast settings (§4.2).

2 BACKGROUND AND MOTIVATION

2.1 First Mile in Live Video Streaming

As mentioned in §1, an E2E live streaming pipeline consists of the ingest and distribution paths (Figure 1). The E2E QoE of live streaming is fundamentally constrained by a single video stream delivered over the first-mile ingest path. First, the quality of the video delivered on this path to the video server imposes an upper limit on the quality of the ABR tracks created from it, and therefore on the QoE of the viewers of the live stream. Second, a player can only download a video segment after the corresponding video content is uploaded to the video server, imposing a latency dependency.

The broadcasting app, a critical component on the ingest path, usually comes in three different forms [66]:

(1) Third-party broadcasting app: There exists standalone software that captures and transmits videos to commercial live streaming services. For example, Open Broadcaster Software (OBS) [48] is a popular broadcasting app that supports live streaming to many commercial services, which highly recommend the use of it [13, 51, 67]. The widely used OBS software supports RTMP (Real Time Messaging Protocol [3]), which is also one of the main protocols that commercial video services use.

(2) Browser-based broadcasting app: Many services such as Facebook and Youtube provide GUIs to open cameras to capture and stream real-time content from their web pages [12, 65].

(3) Mobile-based broadcasting app: Instead of using the browser, smartphone users may prefer the service's mobile app, which also includes GUIs to open the camera and stream videos [11, 64].

Many live broadcasts are originated from mobile devices and transmitted to remote servers using available connections such as Wi-Fi or cellular. To understand how well today's mobile uplinks support the needs of live video ingest, we measure the uplink bandwidth relative to the live ingest bandwidth requirements of commercial broadcasting apps. Specifically, we conduct uplink throughput measurements by uploading a large file over multiple LTE networks, covering various scenarios involving different movement patterns, signal strengths, and locations. Our measurements indicate that the uplink bandwidth exhibits significant variability and can be lower than the sustained bandwidth requirements of commercial broadcasting apps (~1–4Mbps for many live streaming systems). As an illustration, for each of the ten traces that we collected, the 5^{th} percentile of the bandwidth values was less than 2.5Mbps. For four traces, the median observed uplink bandwidth was less than 2.1Mbps. While 5G is expected to further improve the bandwidth, the technology is not yet widely deployed, and has its own challenges (*e.g.*, directivity and sensitivity to blockage for mmWave [37]). Therefore, it is important for applications that need to use uplink cellular connections to be designed appropriately.

2.2 Design Goals

A sound measurement system for ingest path analysis should be able to meet the following requirements.

G1 Enable interested entities such as a testing service or a network operator (who usually do not have access to the detailed design

of, or the source code for the software) to conduct third-party measurements of the performance of a live streaming system.

G2 Be generally applicable to different live video broadcast and distribution platforms instead of targeting a specific setup.

G3 Enable controlled and repeatable experiments.

G4 Be capable of measuring performance dynamics at a fine timescale and enable a tester to holistically reason about the design of the live ingest pipeline.

2.3 Limitation of Existing Analysis Approaches

Existing live streaming analysis tools have several limitations. *First, they conduct limited ingest analysis.* [31] focuses on the distribution path, *e.g.,* instead of directly measuring the video quality (§3.4), it measures the end-viewer perceived video resolution, which characterizes the downlink ABR performance instead of the ingest performance. As §5 will show, the same resolution (ABR track) can have very different quality due to the quality difference of the video delivered on the ingest path. [44, 45] measure the overall E2E QoE instead of for the ingest path, making it difficult to distinguish performance issues on the ingest and distribution paths. They measure different QoE metrics separately under different settings, making it hard to correlate one metric with another. Besides, they mainly focus on the overall session-level QoE instead of its fine-grained dynamics over time. [54] focuses on measuring the delay aspect of the QoE. Therefore, they are not able to meet **G4**.

Second, existing approaches are either broadcasting app-specific or streaming service-specific and hard to generalize. [31] only focuses on streaming from OBS, which provides a user interface to stream local video files, making it hard to generalize to different commercial broadcast platforms such as browser-based and mobile-based broadcasting apps. [44, 45] and [54] rely on service-specific APIs to measure the QoE. As a result, they fail to achieve the aforementioned **G2**.

*Third, they do not meet the **G3** requirement for controllable and repeatable measurements.* [45] and [54] watch online live streams broadcast by other people, having no control of the video source and network conditions. [44] shoots videos playing on a laptop screen as the source, but it suffers from distortion and lighting issues. Although [31] can guarantee the same input video source, it is hard for it to control the network conditions under which its data are collected due to its "in-the-wild" nature.

2.4 Challenges

Achieving the goals mentioned in §2.2 is not straightforward. In addition to the live nature of the video content, the live streaming pipeline is complex and heterogeneous. This creates the following challenges that Livelyzer needs to address:

Complex pipeline. Live video ingest involves a complex pipeline. A broadcasting app captures a video from a camera, encodes it using a codec with an encoding rate control[1] scheme, and transmits a sequence of video frames to the remote server over some network connection. To cope with time-varying network conditions, a broadcasting app may dynamically adapt its upstream transmission.

[1]In video coding, rate control means what an encoder does to determine how many bits to spend for each frame to reach a target bitrate or quality level for the video.

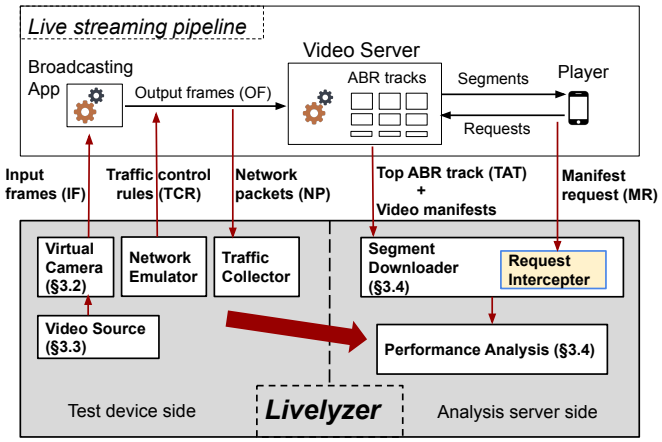


Figure 2: The system architecture of Livelyzer.

There can be different ways of adjusting the amount of data to send to the remote server, *e.g.,* dropping frames or reducing the encoding bitrate. The video server transcodes uploaded video frames into ABR segments. Each of the above components plays an important role and can impact the E2E QoE. This complexity makes it challenging to identify, exercise, and understand the key pieces that impact the QoE.

Heterogeneous design. Different services can have very different designs. Even for the same service, the broadcaster can choose different broadcasting apps of different designs, such as the service's web page in a browser, its mobile app, or a broadcasting app from third parties such as OBS. Unlike the distribution path where HTTP Adaptive Streaming (HAS) is the predominant approach, there is no single *de facto* delivery solution on the ingest path. In fact, there exists a wide range of ingest solutions (*e.g.,* RTMP [3], WebRTC [10], FTL [23], DASH-IF Live Media Ingest [19], *etc.*) with varying levels of publicly available specifications. This makes achieving G2 hard for the ingest path.

Proprietary nature of systems. Live streaming systems usually run proprietary closed-source software. A third party typically does not have visibility into the source code of broadcasting apps and video servers. The uplink network traffic is also usually encrypted, *e.g.,* broadcasting apps may use RTMPS [26] or WebRTC with DTLS [42]. In addition, the increasing use of SSL pinning in mobile applications [44] renders MITM proxy-based approaches increasingly unusable.

Live nature of content. Unlike video on demand (VoD), where the same content can be replayed across multiple experimental runs, live streaming contents are generated in real time. This makes the task of repeating experiments using the same source (**G3**) difficult.

Need for suitable performance metrics. An end user watches a video that flows over both the ingest and distribution paths. While there are well-defined QoE metrics (*e.g.,* quality, stall ratio, startup delay) for the distribution path, there are no well-defined or widely accepted performance metrics for the ingest path. In §3.4 we shall define such metrics of interest.

3 THE LIVELYZER MEASUREMENT SYSTEM

We build Livelyzer, a holistic measurement system that comprehensively examines the ingest performance of different broadcasting apps streaming to various services. As shown in Figure 2, Livelyzer interacts with the live streaming pipeline by generating video source contents and uplink traffic control rules (TCR), monitoring the upstream network packets (NP), and collecting ABR track and manifest information. Livelyzer consists of components running on a test device and an analysis server. The test device hosts different broadcasting apps and part of our software that annotates source videos (§3.3), injects them to broadcasting apps (§3.2), automates measurement tasks, and sends local measurement data to our analysis server after measurement sessions. The analysis server runs the rest of our software, which downloads the top ABR track (TAT) and video manifests, and later analyzes the ingest performance offline (§3.4).

3.1 Black-box Testing

Ideally, we would like to have visibility of every internal point on the ingest path, *e.g.,* the encoder output frames of the broadcasting app (OF in Figure 2) and the application data in the network upstream. However, as mentioned in §2, commercial streaming services and broadcasting apps are usually closed systems. Hence, it is hard to access either the ingest endpoint of the video server to gain visibility into the uploaded frames or the encoder of the broadcasting app to examine the compressed frames.

Given the lack of such internal visibility, we adopt a black-box analysis approach. Specifically, we control both the input (*i.e.,* the video content) and the ingest components (*e.g.,* broadcasting app, streaming service, network condition, *etc.*), and observe the output (*e.g.,* quality of TAT). By changing the input or/and the ingest settings (*e.g.,* through issuing different TCR), we can observe how the output would be affected, and reason about the ingest components.

Our method can test a specific broadcasting app streaming to a particular service under various network conditions in a controlled manner. Specifically, we use a network traffic control tool like Linux `tc` to replay different network conditions based on real-world network traces we collect. Livelyzer runs a traffic collector module on the test device to collect packets on the ingest path. Livelyzer also runs a segment downloader that collects video server output for performance analysis (§3.4). For scalable testing, the broadcast and playback processes are automated (*e.g.,* through Android UI automation and Selenium [43] for browser automation).

3.2 Virtual Video Capture Function

As mentioned in §2, we need to do measurements in a repeatable way. This translates to two requirements: *First*, we need to be able to feed the same video content to different broadcasting apps. *Second*, we need to be able to provide the same video content to the same broadcasting app across different runs. The first requirement comes from our need to compare different broadcasting apps and streaming services fairly. The second requirement is because, for the same (broadcasting app, streaming service) setting, we may want to vary a factor (*e.g.,* network condition) over different runs and keep other factors including the video source the same to examine the sole impact of this specific factor on the ingest performance.

One way to address this is to capture the same scene using a real camera. This approach has several issues: First, it is difficult to provide the same physical scene multiple times in the real world where time and space cannot be reverted. Second, even if we can provide a "repeatable" physical scene like [44] that plays a pre-recorded video on another screen, this still makes it hard to keep the captured video the same due to lighting-related dynamics. Furthermore, the video captured by the camera can contain frames that are a composite of multiple consecutive source frames in the pre-recorded video. This can be caused by the exposure time of the camera capture process lining up with the display times of those frames. Such composites can cause harmful interactions with other components in Livelyzer, such as source annotation (§3.3).

Some broadcasting apps such as OBS support local file input, but not every broadcasting app supports this mode. For example, browser-based broadcasting apps only support camera or screen sharing mode, mobile-based broadcasting apps such as the Facebook and Instagram apps only support camera mode. Screen sharing also introduces non-deterministic distortions in the screen recording process [58], making the video captured by a broadcasting app (*i.e.,* the recorded screen) just an approximation of the source video.

To provide a universal interface to commercial broadcasting apps for capturing repeatable video contents, we create a virtual camera in Livelyzer. It takes a local video file as input and behaves like a normal camera device from the perspective of a broadcasting app. The video source can be fed into the virtual camera at different frame rates. The virtual video capture function extracts the sequence of frames from an input video file, and redirects them to the virtual camera. The broadcasting app then captures input frames (IF in Figure 2) by sampling these raw frames based on the broadcasting app's frame rate setting. In Linux, a virtual camera can be realized using `/dev/videoX`. We leverage `v4l2loopback` [52] to create such a new virtual video device. To capture video with it, we use `FFmpeg` [15] to specify a local video file as the input and the virtual video device as the output.

3.3 Crafting Video Source Files

Given the virtual camera, we still need to prepare the input video content. To measure both frame loss and quality of delivered video on the ingest path, we need to associate each frame in the received video with its corresponding frame in the source.

Achieving frame alignment in live video ingest is challenging because the frames in the source and received videos are not naturally aligned for several reasons. First, different broadcasting apps could capture videos at different frame rates. Second, during transmission to the remote video server, a broadcasting app may drop frames to adapt to varying network conditions. Third, depending on when it joins the live event, a player will not necessarily start playing a video from the beginning of a broadcast. Therefore, the first frame being played may be different from the first frame captured by the broadcasting app. Also, depending on the extent of time synchronization between when the camera is turned on and when the broadcaster starts streaming to the remote video server, even the first frame captured (to be encoded) by the broadcasting app may be different from the first frame that is seen by the recording camera.

One approach for achieving frame alignment would be to compare every frame in the received video and every frame in the source based on content similarity. However, the challenge with this approach is that in the case of a static or a repeating scene, multiple frames in the source would be visually similar to each other. In this case, a frame in the received video may be mapped to multiple frames in the source, making unambiguous frame alignment hard.

To achieve frame alignment, we overlay a unique signature to every source frame and leverage computer vision techniques to detect the signatures from frames in the received video to match each of them to a source frame. The signature should be robust to compression artifacts (*i.e.,* be still recognizable from low bitrate streams created under poor network conditions). We leverage the Quick Response (QR) code [57] to create this specific signature since it is more robust than the approach of overlaying a sequence of digits (*e.g.,* a frame number) due to the QR code's use of Reed–Solomon error correction [56]. We empirically verified this when we were researching suitable signature technologies in the early phase of our study. In addition, we pad the source video with dummy frames before and after the original video content to make sure that the original video content gets captured and eventually played regardless of the level of time synchronization among different components.

3.4 Analyzing Ingest Performance

As mentioned in §2, we need to define performance metrics on the ingest path that impact end-user QoE. The metrics we consider are video quality, effective frame rate, ingest freshness, and ingest smoothness (to be detailed shortly). To facilitate ingest performance analysis, we need to measure the quality and timing of the stream delivered to the video server. However, extracting video frames in such a stream on the ingest path is challenging (§2.4). Instead, we use the transcoded TAT to approximate the above stream, easily accessible using standard HTTP APIs from the distribution path. TAT is also more closely associated with end viewers' experience as players ultimately need to fetch video segments from ABR tracks, and TAT represents the best quality encoding any user can receive. Livelyzer runs a segment downloader on its analysis server to fetch ABR tracks. We parse the corresponding live video manifest for the session to obtain the address information for the TAT, which is passed to our segment downloader for downloading the TAT segments. During the live stream, the manifest is periodically updated over time as new segments are generated at the server. Our segment downloader fetches these updated versions at regular intervals to extract the information required for computing the various performance metrics described next.

Video quality. We examine the quality of video segments in the top ABR track (TAT) created by the video server. TAT represents an upper bound on the video quality experienced by any user served by the video server. The video quality of the TAT depends on the quality of the content received by the server on the ingest path, which is the cumulative end result of all ingest activities (including broadcasting app encoding and adaptation, network performance, *etc.*).

For the specific video quality, we adopt Video Multimethod Assessment Fusion (VMAF) [27, 29, 30], which is a recently proposed perceptual quality metric and has been shown to perform much better than traditional video quality metrics that do not accurately capture human perception, such as Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM) [55]. VMAF is a full-reference model allowing us to measure the perceptual quality of a distorted video by comparing it with regard to a pristine quality reference of the same content. VMAF was originally designed for evaluating compression artifacts where the reference and distorted videos have the same frame rate. In contrast, in the live video ingest use case, the source video and TAT can have different frame rates (see §3.3). Therefore, we first resample the TAT to match its frame rate with the source video using FFmpeg [15]. We then use the frame alignment step (§3.3) to align the first frame in the TAT (t1) with its corresponding frame in the source (s1). Then we calculate VMAF using the sequence of source frames starting from s1, as the reference sequence, and the sequence of TAT frames starting from t1, as the distorted sequence. Since we are particularly interested in the quality of the content consumed by mobile users, we use the VMAF phone model (designed for small screens [28]) for VMAF calculation. Since we are interested in measuring video quality at a fine granularity over the video session, we use the arithmetic mean of the VMAF values of all the frames in a segment as the segment's VMAF value, and analyze the distribution of per-segment VMAF values [2, 40] across the session. Note that while we use VMAF for the reasons stated above, Livelyzer can easily accommodate other video quality metrics, *e.g.,* PSNR and SSIM.

Effective frame rate. Broadcasting apps may capture frames at a different frame rate (FPS) and drop frames when the network bandwidth becomes insufficient. Besides, networks may drop frames, and video servers may reduce frame rates as well. To quantify the impact of frame loss, we define effective frame rate (effective FPS, or eFPS), the number of *distinct* frames in each second in TAT. eFPS equals FPS when there are no duplicate frames (*i.e.,* every frame is distinct). However, according to our observations (§5.1.1), video servers may duplicate frames to maintain a constant FPS in ABR tracks when the frame rate on the ingest path is variable. Therefore, to compute eFPS, we consider only distinct frames by identifying and removing duplicates using our frame annotation and alignment methods (§3.3).

Ingest freshness. We are also interested in understanding the latency impact of the ingest path, which affects the E2E broadcaster-to-viewer (B2V) delay (§2.1) – a measure of how much a viewer is behind the live event. To characterize ingest freshness, we define the *ingest delay* for each ABR segment as the time elapsed from when its first frame is generated at the source to when all the ABR track ladder variants of that segment become available at the video server for players to download. This segment-level delay is also more related to end users' experience compared to the frame-level delay — a segment[2] becomes available for players to download only after all the frames in the segment arrive at the server and the different ABR track variants for that segment have been created. The ingest delay for a segment is the sum of the times spent on broadcasting app encoding, network transmission, and server transcoding. The ingest delay is part of the E2E B2V delay. Therefore, a longer ingest

[2]In this paper, we use segment to refer to the smallest unit of data that can be requested by an end viewer, *e.g.,* an ABR segment or a CMAF [17] chunk.

delay will lead to a longer E2E B2V delay, everything else remaining the same. For each new updated version of the live manifest, we extract the time t_s that it was updated at the server. We mark the arrival time as t_s for all ABR segments that first appear in the latest version of the manifest and were absent in earlier versions. The generation time (t_b) of the first frame of each segment is recorded at the virtual camera. The ingest delay, or broadcasting-app-to-server (B2S) delay of a segment, can thus be calculated as $t_s - t_b$.

Ingest smoothness. Once a viewer joins a live event, in order to play a live stream smoothly without stalls, the player needs to get subsequent ABR segments from the video server in a timely fashion. This requires the ABR segments to be created and made available for downstream players in a timely manner. However, during a live stream, if the ingest delay increases significantly (*e.g.,* because the uplink bandwidth become very low for some time), the arrival of the video frames at the video server and subsequent creation of the corresponding ABR segments will be delayed, increasing the chance that a player may not receive some segments in a timely fashion and therefore experience stalls. While due to live streaming freshness considerations, a player cannot stay far behind the live edge, for many common use cases, the player can still start several seconds behind the live edge (we define it as *offset*) and so can tolerate some variability in the availability time of the segments. Here, to measure the effect of this variability on user experience, we assume a player with an X seconds offset behind the live edge (we empirically set X = 10 in our experiments), and measure the stall behavior due to segment availability time variability. We define the stall ratio to be the aggregate stall duration as the percentage of the total live video session duration.

4 USING LIVELYZER FOR LIVE VIDEO ENCODING ANALYSIS

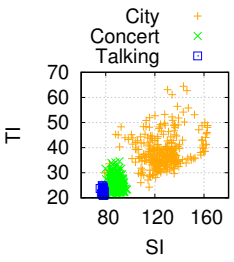


Figure 3: Content complexity measured by spatial information (SI) and temporal information (TI).

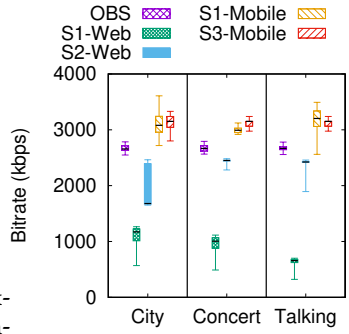


Figure 4: Encoding bitrate measured with different videos.

We next use Livelyzer to characterize the video encoding design on the live streaming ingest path, spanning broadcasting apps' encoding and video servers' transcoding, both important QoE-impacting components in the E2E live streaming pipeline (§2.1). We first consider high network bandwidths scenarios to ensure that the observed video outputs of the two components reflect their inherent application logic and are not caused by any network bandwidth limits. We shall later use this behavior as a baseline when we explore more bandwidth-constrained situations in §5.

We use three different representative video contents as our broadcast sources: (1) City – a city view with a lot of detail, (2) Concert – a live show that involves significant movement, (3) Talking – a talking person with an almost static background. The videos are obtained from Youtube in 1080p and 30fps. We select a 5-minute long sample of each for this study. We stream the 720p variant of the content (obtained by downscaling the 1080p reference to 720p) as input to broadcasting apps, in line with industry recommendations [14], and use the original 1080p version as the pristine quality reference when computing VMAF (§3.4). Figure 3 depicts the spatial and temporal information [1], commonly used to characterize scene complexity, for these videos. Each data point in Figure 3 represents an individual segment from the corresponding video.

The three commercial services we examine are denoted as *S1*, *S2*, and *S3* in the rest of the paper. We use the terms Web and Mobile to refer to the browser-based and mobile-based broadcasting apps, respectively, for each service.

4.1 Encoding Design of Broadcasting Apps

We first study the encoding bitrates of videos created by different broadcasting apps covering common broadcast use cases (§2.1). As mentioned earlier, experiments in this section are conducted under high-bandwidth network conditions, so the encoder should not be constrained by any bandwidth concerns[3]. This scenario represents the best-case performance of commercial broadcasting apps - we shall use it as a baseline when studying the rate adaptation performance of these systems under real-world network conditions when the upstream network bandwidth is not plentiful and is time-varying (§5).

Figure 4 shows the encoding bitrate distributions for different contents encoded by different broadcasting apps. We measure these values from the uplink network traffic by computing the data sending rates over time. Since we cannot extract the precise Group of Pictures (GOP) structure used by the different commercial broadcasting apps (*e.g.,* due to traffic encryption), we use 6 seconds as the interval to compute each bitrate sample. For OBS, we only present its encoding bitrate distribution when streaming to *S1* as the results for streaming to *S2* and *S3* are very similar.

We make two main observations. First, different broadcasting apps have very different outputs with different encoding bitrate distributions even for the same content, suggesting different encoding settings, *e.g.,* mobile-based broadcasting apps tend to use consistently higher bitrates than OBS and browser-based broadcasting apps.

Second, different broadcasting apps have very different encoding rate control behaviors. The OBS encoding bitrate is tightly concentrated around 2.7Mbps across all the content, suggesting the use of a CBR encoding independent of the content type. *S1-Web*, however, produces encodings with average bitrates and bitrate spreads that differ for different content – this is more consistent with VBR-like encoding behaviors. Other broadcasting apps, such as the mobile-based broadcasting apps, exhibit modest bitrate variations: the bitrates are within 18% and 8% of the corresponding mean values for *S1* and *S3*'s mobile-based broadcasting apps, respectively.

[3]We leverage our source video padding (§3.3) to ensure that the bitrate has already "ramped up" when the actual video content comes.

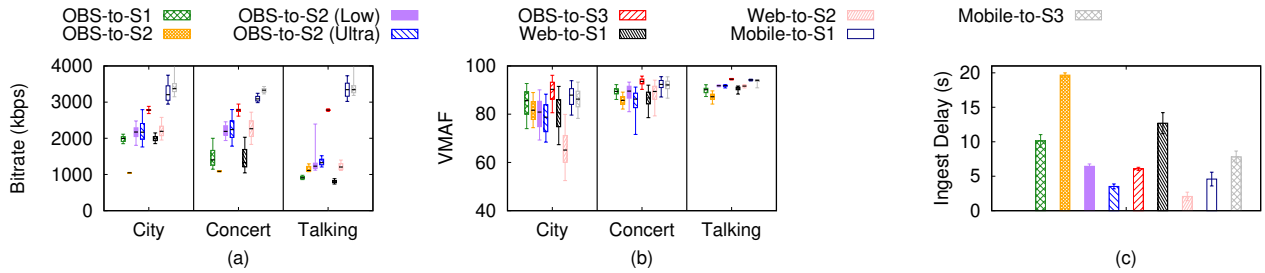


Figure 5: (a) Bitrate distribution of TAT. (b) Quality of TAT. (c) Broadcasting-app-to-server delay of segments in TAT.

They exhibit similar bitrate spreads for encoding the three different video contents.

Above, we observe that many broadcasting apps we study use CBR. Given that VBR has advantages over CBR, such as being able to achieve higher video quality with the same average bitrate or provide the same quality with a lower bitrate encoding to better accommodate to bandwidth constraints [40], one potential research direction is to explore using VBR encoding in broadcasting apps.

4.2 Server ABR Transcoding Design

We now study how different services transcode a received live video stream into ABR tracks. We focus on the top ABR track (TAT), which represents the highest-quality stream that end viewers could enjoy. Any quality impairment observed in this track is entirely caused by the ingest component. We measure the bitrate, duration, frame rate, and ingest delay of each segment in this track.

Figure 5(a) shows the bitrate distribution of the TAT created by different services' remote video servers. For OBS streaming to *S2*, we include three modes that *S2* provides for RTMP ingest: default normal mode, low latency mode (denoted as Low), and ultra-low latency mode (marked as Ultra). As shown, different video servers also have very different encoding (transcoding) bitrate distributions for encoding the same content, similar to different broadcasting apps do (§4.1). However, here the content dependency compared to broadcasting apps' encodings appears to be higher overall: many video servers (*e.g., S1*'s video server receiving streams from OBS and *S2*'s video server receiving streams from *S2-Web*) use fewer bits to encode less complex content such as the "Talking" video. Also, we observe that the "dependence" between broadcasting apps' encoding bitrates and the ABR transcoding bitrates differ among services. For example, OBS encoding is very CBR-like (§4.1) while the corresponding *S1* video server creates content-dependent VBR encodings. However, for OBS streaming to *S3*, the bitrate distributions of OBS encoding and that of *S3* server transcoding are very similar (both centered around 2.7Mbps).

Table 1 shows the duration and frame rate of segments created by different servers. As shown, the segment duration can be very different depending on the service and broadcast platform. Later we shall see how the segment duration affects the ingest delay.

4.3 QoE Impact

We next examine how different encoding rate control schemes in broadcasting apps and video servers affect the end viewer QoE.

Table 1: Comparison of server ABR transcoding design.

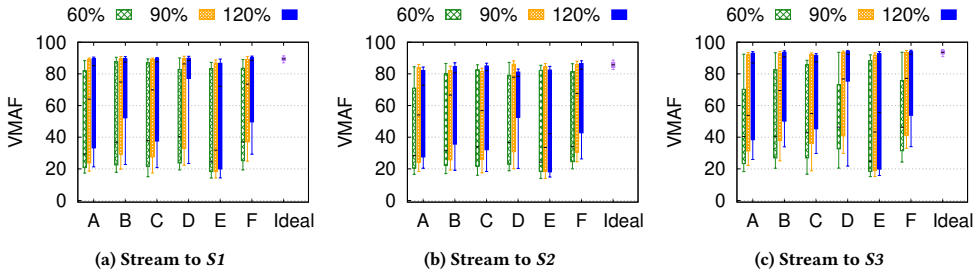
Streaming service	Broadcasting app	Mode	Segment duration	Frame rate
S1	OBS	N/A	2s	30 fps
	S1-Web	N/A	2s	30 fps
	S1-Mobile	N/A	2s	24 fps
S2	OBS	normal	5s	30 fps
		low latency	2s	30 fps
		ultra-low latency	1s	20-30 fps
	S2-Web	N/A	1s	20-30 fps
S3	OBS	N/A	2s	30 fps
	S3-Mobile	N/A	5s	30 fps

Figure 5(b) shows the video quality of TAT created by each video server. Usually, a change of 6 or more VMAF points would be noticeable to a viewer. We observe that even under unconstrained uplink network conditions, the video quality is not always high. The "Talking" video has a much higher quality than the other two videos, likely a result of its relatively lower content complexity. Overall, settings with both high encoding bitrates at the broadcasting app and server sides (*e.g., S1-Mobile* and *S3-Mobile*) also have a high video quality.

Figure 5(c) shows the ingest delay (§3.4). Overall we can see a correlation between the delay and the segment duration shown in Table 1: the larger the segment duration, the higher the ingest delay even though the broadcasting app is not necessarily doing segmented delivery to the server. The reason for the correlation is: (1) The ingest delay of a segment covers the time between when its first frame is generated at the source and when its last frame is uploaded to the ingest server, which depends on the segment's duration, (2) transcoding a larger segment usually takes longer than transcoding a shorter one. We also observed differences across services. For instance, both *S2* (normal mode) and *S3-Mobile* use 5s as the segment duration, while *S3-Mobile* has a much shorter ingest delay than *S2*.

5 USING LIVELYZER FOR NETWORK RATE ADAPTATION ANALYSIS

Next, we study how different broadcasting apps adapt their upstream transmission to cope with network dynamics, and the resulting impact on performance. As mentioned in §2.1, there can be different ways of adjusting the amount of data to send to the



(a) Stream to *S1* (b) Stream to *S2* (c) Stream to *S3*

Figure 6: Video quality of OBS streaming to different services: each network trace (A-F) is scaled to 60%/90%/120% of the baseline video encoding bitrate

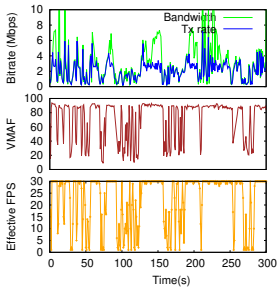
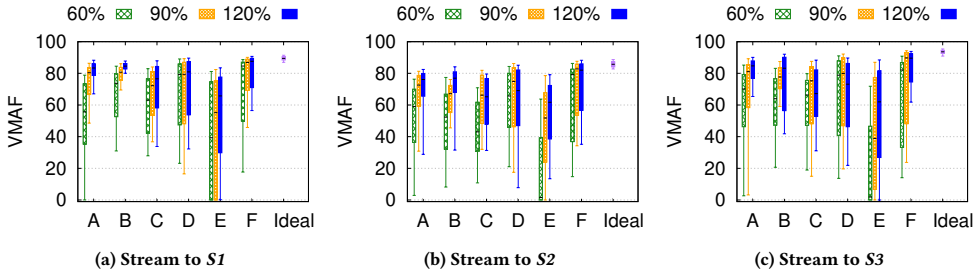


Figure 7: OBS drops frames to adapt to changing network conditions.



(a) Stream to *S1* (b) Stream to *S2* (c) Stream to *S3*

Figure 8: Video quality of OBS dynamic bitrate mode streaming to different services: each network trace (A-F) is scaled to 60%/90%/120% of the baseline video encoding bitrate

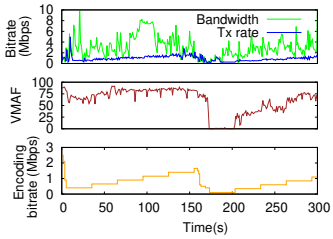


Figure 9: OBS-dynamic increases encoding bitrate slowly when bandwidth increases.

network, *e.g.,* dropping frames or reducing encoding bitrates. To understand the adaptation behavior, we measure the performance evolution across time and correlate it with the corresponding prevailing network bandwidth condition.

We focus on the "Concert" video with a medium content complexity across the three videos we have (Figure 3). We leverage the network condition emulation feature of Livelyzer (§3.1) to replay six real-world network bandwidth traces. For each of the six cellular uplink traces (denoted as A, B, C, D, E, and F, whose coefficients of variation are 1.11, 0.90, 0.84, 0.65, 1.08, and 0.69, respectively), we create three variants as follows for each broadcasting app. We proportionally scale the per-second bandwidth values in a trace such that the average bandwidth of the resulting scaled trace becomes either 60%, 90%, or 120% of the average of the encoding bitrate time series output by that app under plentiful uplink network conditions (§4.1). This, in total, creates $6 \times 3 = 18$ different network conditions for replay for each broadcasting app.

5.1 Using Third-party Broadcasting App: OBS

We start with understanding the third-party OBS broadcasting app. We consider both the default OBS and OBS with dynamic bitrate adaptation mode enabled (denoted as OBS-dynamic). Starting with OBS version 24 back in 2019, "dynamic bitrate mode" was added as an optional scheme to replace the default rate adaptation scheme. We use these two schemes to stream videos to *S1*, *S2*, and *S3*. For *S2*, we focus on its default normal streaming mode (§4.2).

5.1.1 Default OBS. Figure 6 overviews the quality of the video streaming from OBS to different services under the 18 different network conditions described above. For comparison, we also show the baseline video quality under ideal network conditions measured in §4. In general, the higher the average bandwidth, the higher the video quality we observe. The instantaneous network bandwidth and its variability over time also also have an impact on video quality. For example, trace D has the lowest coefficient of variation, leading to a relatively better video quality than others. *Overall, the VMAF values show considerable degradation compared to the results under no bandwidth constraints (§4.2): the (25^{th} percentile, median) VMAF averaged over the 18 network conditions is decreased by (64%, 31%), (62%, 28%), and (66%, 34%) compared to the baseline for S1, S2, and S3, respectively.* Besides, we still observe low quality especially in the tail of the distribution (*e.g.,* 5^{th} and 25^{th} percentiles of the VMAF distribution) even when the average network bandwidth for a network condition is high (*e.g.,* 120% of the encoding bitrate). The average stall ratio (defined in §3.4) across different traces is 3.5%, 1.0%, and 2.1% for *S1*, *S2*, and *S3*, respectively (not shown in the figures).

To understand the root cause for such performance degradation, we plot the evolution of network bandwidth, data sending rate, video quality, as well as the effective frame rate of a typical live video ingest session across time, as shown in Figure 7. This figure shows an example of how OBS performs when the upload bandwidth availability changes over time according to one trace. As shown, there are still many low-bandwidth periods even when the average

bandwidth is high (120% of the encoding bitrate in this case). In high-bandwidth periods, even if the bandwidth is higher than the baseline encoding bitrate, due to the real-time nature of the live video stream, the additional bandwidth availability cannot be used to further improve the video quality. We also see a relationship among the network bandwidth, VMAF, and the effective frame rate (effective FPS, defined in §3.4): when the network experiences low-bandwidth periods, the effective FPS in the top ABR track (TAT) is very low (e.g., zero), leading to low VMAF values and choppy video quality. The measured low effective FPS is likely because some frames get dropped before reaching the ABR server.

To further explore the above, we instrument the OBS source code[4] to collect frame management information. We find that *the default OBS broadcasting app drops frames when the network bandwidth becomes insufficient to support the configured video encoding bitrate. We also note that the frame drop process is bursty – sequences of consecutive frames are dropped. Some bursts can be as large as 2s worth of frames, leading to poor video experience during that time.* We also examined the TAT created by different streaming servers. We find that *S1* and *S2* duplicate frames to maintain a constant high frame rate, although the effective frame rate (§3.4) still remains low. *S3* adopts a different strategy – it does not fill the gaps in the sequence with duplicate frames and uses discontinuous presentation times (PTS) to indicate the presence of gaps so that the player knows when to render each frame.

5.1.2 OBS dynamic bitrate mode. We now study the performance of this new mode under the same network conditions as in Figure 6. Overall, the video quality is improved compared to the default OBS rate adaptation (see Figure 8). However, there are some scenarios when the quality becomes worse, such as when using trace E with its average bandwidth scaled to 60% of the encoding bitrate. The stall ratio is 4.3%, 0.8%, and 2.1% for *S1*, *S2*, and *S3*, respectively.

To understand why OBS-dynamic sometimes has worse performance than the default OBS adaptation, we examine the different performance metrics across time. Figure 9 shows an example run for this broadcast setting. We find that the network resource frequently becomes under-utilized: even if during many periods the bandwidth is higher than the baseline OBS encoding bitrate, the broadcasting app does not leverage the increased bandwidth to reach the baseline bitrate. The VMAF can stay low even after the network bandwidth rapidly improves from a low value, *e.g.*, even if the bandwidth is only very low at a few points, the near-zero poor VMAF lasts more than 30s.

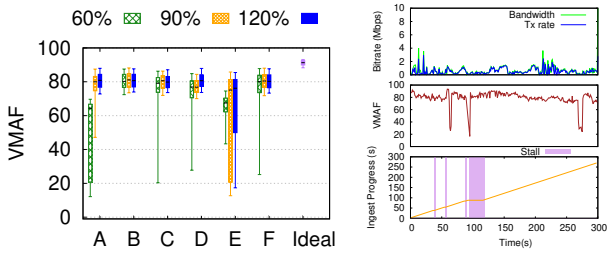
To understand the root cause for the above behavior, we further instrument the corresponding dynamic bitrate adaptation module of OBS. Specifically, we log the encoding bitrate decision over time, as shown in the bottom subfigure of Figure 9. We can see that the encoding bitrate increases relatively infrequently (every ~30s), and it increases very little at each step. By examining its source code, we find that although OBS-dynamic reduces its encoding bitrate whenever the network condition degrades from good, it only increases the encoding bitrate very gradually over time, when the network connection starts recovering from a poor bandwidth condition. Specifically, it reduces the encoding bitrate when the measured frame buffer occupancy is high and sets the new encoding bitrate to the buffer drain rate, which approximates the available network bandwidth. When the current encoding bitrate is lower than the baseline bitrate, the scheme only checks whether it is safe to increase encoding bitrate every 30s based on the frame buffer occupancy: if the buffer occupancy is low, it would increases the encoding bitrate. Worse, every time OBS-dynamic decides to increase the bitrate, it only increases the bitrate by a fixed amount ($\frac{maxBitrate}{10}$) and keeps probing until it reaches $maxBitrate$, which is the default encoding bitrate specified by the system/user, regardless of the current network condition. *As a result, even when the network condition already becomes good right after a temporal outage at around 170s, it takes OBS more than 100s to fully recover to the default encoding bitrate under high network bandwidth conditions.* In §6.1, we show how this rate adaptation logic can be improved by demonstrating our proposed rate adaptation scheme.

[4]Unlike many other broadcasting apps, OBS is open-source

5.2 Using Browser-based Broadcasting Apps

We now study the browser-based broadcasting apps, specifically the *S1-Web* and *S2-Web* broadcasting apps. *S3* does not support sending streams from a browser and requires either a third-party broadcasting app or the service's own mobile app for broadcasting.

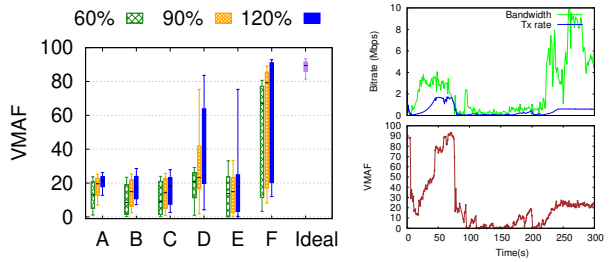
5.2.1 S1-Web. As shown in Figure 10a, under many settings, *S1-Web* still has high quality video encoding despite network bandwidth constraints. However, this leads to high stalls, as shown in Figure 10b, an example run under trace B scaled to 60% of the original video encoding bitrate. The average stall over different traces is 11% – the highest stall ratio across different broadcasting apps that we studied. This suggests that *S1-Web* tends to maintain a relatively high data sending rate, which can overshoot the network when the bandwidth is limited, leading to a high stall ratio despite maintaining a relatively high video encoding quality.



(a) Quality distribution (b) Example run

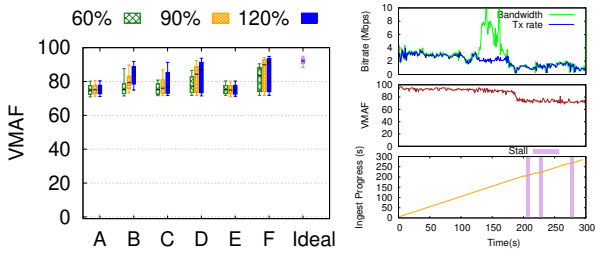
Figure 10: Browser-based broadcast to *S1*.

5.2.2 S2-Web. *S2-Web* has very different behavior from *S1-Web*. As shown in Figure 11a, most of the scenarios exhibit low video quality, *e.g.*, even the 95^{th} percentile VMAF values of the first six settings are all less than 30. On deeper exploration, we find that once the bandwidth increases after a period of drops, *S2-Web* still keeps sending data at a low rate instead of increasing it to the baseline data rate (~2Mbps, §4.1), as exemplified by Figure 11b. The average stall ratio (0.7%) is much lower compared to *S1-Web*.



(a) Quality distribution (b) Example run

Figure 11: Browser-based broadcast to *S2*.



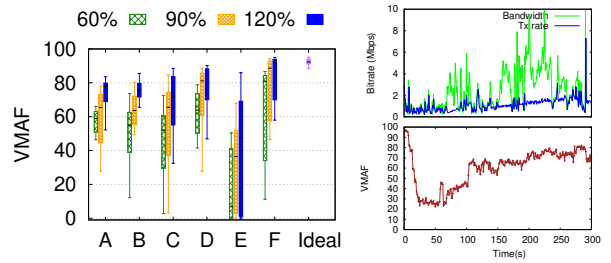
(a) Quality distribution (b) Example run

Figure 13: Mobile-based broadcast to *S3*.

5.3 Using Mobile-based Broadcasting Apps

We next study the *S1* and *S3* mobile app broadcast performances. The *S2* mobile app broadcast feature requires the streaming account to have more than 1000 subscriptions, making it difficult to conduct experiments, and is not studied.

5.3.1 S1-Mobile. Figure 12a shows the video quality distribution. We find that the broadcasting app increases the data sending rate slowly when the network conditions recover, missing opportunities to increase the encoding bitrates and thereby the video quality, similar to the behavior of OBS-dynamic. Figure 12b shows such an example: when the network condition recovers at t ≈ 70s, the broadcasting app's data sending rate increases very slowly, making the VMAF recovery slow as well. The stall ratio is 4.2% on average across different settings.



(a) Quality distribution (b) Example run

Figure 12: Mobile-based broadcast to *S1*.

5.3.2 S3-Mobile. We observe relatively high video quality, as shown in Figure 13a. However, we also observe severe stalls, 6.3% on average. Figure 13b shows an example run where a stall occurs when the network bandwidth drops at t ≈ 180s. The video quality decrease indicates that *S3-Mobile* does adapt to the network bandwidth reduction by reducing the encoding bitrate. But the stall keeps occurring, which suggests that *S3-Mobile*'s rate adaptation is sub-optimal and can be improved (in this case, it still overshoots a little, causing an extra stall at t ≈ 275s).

To summarize, our measurements show a vast diversity of performances across different scenarios, suggesting different broadcasting apps and services choose different points in the design spaces for the live streaming ingest pipeline. This reinforces the need for tools like Livelyzer to analyze systems and understand their performance profiles as well as their strengths and weaknesses.

6 IMPROVING UPSTREAM INGEST DESIGN

It is vital to deliver the content on the first mile at high quality with minimal impairments, as it becomes the source reference used for the ABR track encoding and streaming delivery to end users, and its quality constrains the end-user QoE (§2.1). However, the limited bandwidth and variability on the first mile make this task difficult, as shown by our characterization of existing designs (§5).

In this section, we conduct a what-if analysis to explore the potential of two techniques for improving the video quality on the first mile: (1) suitable rate adaptation to better adapt to network conditions (§6.1), and (2) using more efficient codecs (§6.2).

6.1 Improving Rate Adaptation Logic

For ABR streaming, rate adaptation has been studied extensively on the distribution path from the server to the client, with the latter dynamically selecting from different variants over time. On the ingest path, there is a single variant being transmitted, and any adaptation would involve dynamically changing the encoding bitrate. Conceptually, an adaptation scheme that tailors the video bitrate to the actual network bandwidth variability should be able to deliver better video QoE. However, there has been much less research on the rate adaptation for the live streaming ingest case.

§5 shows that while the existing broadcasting apps seem to have rate adaptation schemes built in, their performances could be further improved. For example, §5.1.2 indicates that when the bandwidth increases from a low level, OBS-dynamic's encoding bitrate always increases by a fixed delta at 30s intervals, irrespective of the actual network bandwidth, leading to suboptimal performance (*e.g.,* Figure 9).

To understand the potential improvements possible with a better adaptation logic, we design a simple proof-of-concept rate adaptation scheme that better adapts to changes in the available bandwidth. Our algorithm works as follows. In every epoch, we compute a new encoding bitrate for the next epoch based on the predicted uplink bandwidth for the next epoch. The new encoding bitrate is calculated as $min\{(1-\eta) \times BW, maxBitrate\}$ where BW is the predicted throughput for the next epoch, $\eta \in (0, 1)$ is a tunable parameter introduced to control the aggressiveness of the adaptation

algorithm in terms of bandwidth consumption, and $maxBitrate$ is the maximum video encoding bitrate specified by the system/user.

We implement this new adaptation logic in OBS the open-source broadcasting app. We denote it as OBS-adapt and compare it with the default OBS and OBS-dynamic adaptation schemes. We use the following bandwidth prediction approach. In our network bandwidth trace-driven experiments, we take the average of the bandwidth values for the past N epochs as the estimated network bandwidth for the next epoch for input to OBS-adapt. We empirically set η to 25%, N to 4, and the epoch to be 2s.

Figures 14, 15, and 16 show the video quality associated with streaming from OBS to *S1*, *S2*, and *S3*, respectively, under the 18 different network conditions described earlier for OBS. As shown, OBS-adapt noticeably improves the video quality across different network conditions. The (25^{th} percentile, median) VMAF averaged over the 18 network conditions and three services for OBS-adapt improves by (29.9, 21.7) and (14.5, 12.1) compared to OBS and OBS-dynamic, respectively. The average stall ratio is similarly low for all three schemes, with OBS-adapt being slightly lower: 2.2%, 2.4%, and 1.7%, for OBS, OBS-dynamic, and OBS-adapt, respectively.

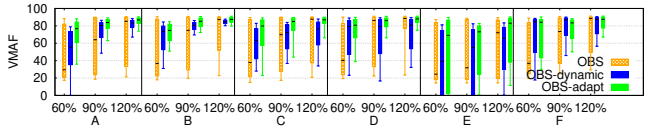


Figure 14: Ingest performance comparison of the default and improved OBS streaming to *S1*.

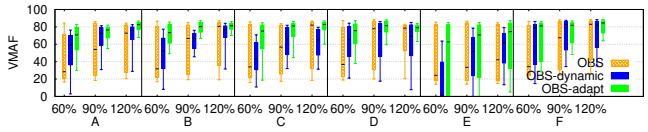


Figure 15: Ingest performance comparison of the default and improved OBS streaming to *S2*.

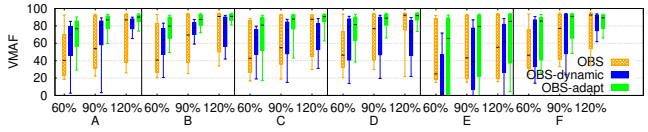


Figure 16: Ingest performance comparison of the default and improved OBS streaming to *S3*.

To better understand the reason for the observed performance improvements, we consider an example experimental run under one bandwidth profile. As shown in Figure 17, OBS-adapt's VMAF values remain high compared to OBS and OBS-dynamic. Although OBS-dynamic also adjusts its encoding bitrate to cope with the network condition, compared to OBS-adapt, it frequently underutilizes the available network resources. Specifically, it selects a much lower encoding bitrate than the available network bandwidth, due to its specific adaptation logic when the available network bandwidth increases from a low value (*e.g.,* at t ≈ 70s and t ≈ 155s). In contrast, OBS-adapt is able to better adapt to the same changing bandwidth conditions, leading to better video quality. The default OBS adaptation scheme drops frames instead of adjusting the encoding bitrate, leading to frequent very low quality periods.

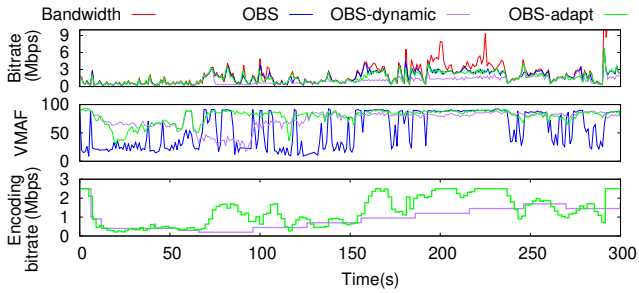


Figure 17: Example run showing the network bandwidth, data rate, video quality, and encoding bitrate decision evolution over time.

The above results clearly show that even a relatively straightforward adaptation strategy that fully accounts for network bandwidth changes can significantly improve the delivered quality on the ingest path compared to the current state of the art, even under challenging network conditions. Note that developing an overall optimized adaptation strategy on the ingest path is not straightforward and involves various challenges, distinct from the adaptation on the distribution path. We leave the development of a full-fledged ingest adaptation strategy to future work.

6.2 Using More Efficient Codecs

We next explore the potential of using more advanced codec technologies with greater compression efficiencies, which require fewer bits to encode the same quality compared to the widely used H.264 to improve the quality of the delivered video on the ingest path.

We conduct a what-if-analysis to answer this question, using H.265 [49] as the example advanced codec. H.265/HEVC has significantly higher encoding efficiency than H.264/AVC [8]. Optimized software and hardware implementations are widely deployed in devices, making it the second most widely used video coding format after H.264 [6]. Existing literature has focused either on comparing the encoder outputs across codecs or on the downstream delivery from the video server to end users. There is little quantitative understanding of the utility of using H.265 instead of H.264 on the ingest path.

To fill this gap, we characterize H.265-enabled live video ingest and compare its performance with the original H.264 counterpart under real-world cellular uplink network conditions. Since popular commercial live video broadcasting apps and streaming services do not yet support H.265[5], we had to set up our own H.265 live video ingest testbed for this analysis. Specifically, we leverage the NGINX-RTMP module [4] on top of the NGINX server as our RTMP

[5]Youtube provides APIs for H.265 with HLS ingest. However, it is not widely used by broadcasting apps yet and can inflate delay compared to RTMP-based ingest.

ingest endpoint. We port the popular x265 codec [53] to OBS and add support for H.265 decoding to NGINX-RTMP, following the H.265 decoder configuration record format specified in ISO/IEC standards [16].

Figure 18 shows the resulting video quality under identical network conditions, when using H.265 compared to H.264 encoding with the OBS broadcasting app with our proposed OBS-adapt adaptation scheme (§6.1). The (25^{th} percentile, median) VMAF score averaged over the 18 network conditions is increased by (17.4, 7.0) points compared to H.264. The average stall ratios under both codec settings are very similar (~1.1%).

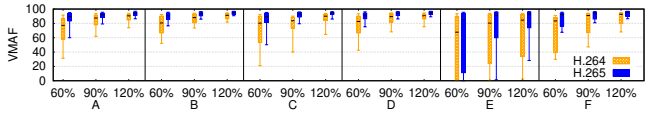


Figure 18: H.264 *vs.* H.265 for OBS-adapt to NGINX.

Recall that in §6.1, we compare different rate adaptation schemes by streaming from OBS with H.264. Now we further study how various adaptation schemes perform under H.265. As shown in Figure 19, our proposed scheme improves the video quality compared to the default OBS and OBS-dynamic under different network conditions. The (25^{th} percentile, median) VMAF score averaged over the 18 network conditions is increased by (49.9, 26.3) points and (8.3, 6.4) points compared to OBS and OBS-dynamic, respectively. The stall ratio is similar for the different schemes: 1.2%, 1.5%, and 1.1%, for OBS, OBS-dynamic, and OBS-adapt, respectively.

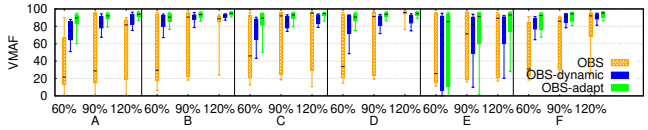


Figure 19: Ingest performance comparison of default and improved OBS streaming to NGINX with H.265.

The above results illustrate that using a combination of an improved codec and a better rate adaptation scheme can help improve the quality delivered from the broadcasting app to the video server. Note that the improved video quality enabled by H.265 comes at the cost of increased coding-related computation overhead. The average broadcast device CPU utilization we measure is 6.2% and 14.0% for H.264 and H.265 streaming, respectively. This is a trade-off that should be considered depending on the use case, *e.g.,* when broadcasting with a smartphone with a limited battery.

7 RELATED WORK

Analyzing live video streaming. There have been recent studies on both the technical aspects [9, 31, 39, 44, 45, 54, 68] and human factors [22, 47, 50] of live video streaming. [44, 45, 54] used service-specific APIs to study Periscope and Facebook Mobile. [31] studied 360-degree live streaming from OBS to Facebook and Youtube. Compared to these studies, our work differs in two significant ways: First, we focus on the upstream ingest path. Second, their methodologies are based on specific service features compared to our more general measurement approach.

Improving live streaming performance. [41] attempted to enhance time-shifted live streaming by slightly sacrificing the QoE of live users to greatly enhance the QoE for VoD users watching the same video later. [35] dealt with improving the CDN systems for live video delivery. [25] proposed the use of super-resolution to save bandwidth on the ingest path. [5] developed downstream bandwidth prediction techniques for ABR adaptation of low-latency chunked live streaming. Low-latency variants, including LL-DASH [18] and LL-HLS [38], have been recently proposed to reduce the latency on the distribution path.

Video conferencing. There have been studies on measuring [61] and improving [20, 69, 70] video conferencing performance. Unlike video conferencing that usually has tens of milliseconds of latency requirements, live streaming can tolerate several seconds of end-to-end latency and usually needs a transcoding server for ABR track creation. As a result, existing studies on video conferencing focus on how to reduce latency. They also assume the use of WebRTC. Unlike these studies on video conferencing, we found that commercial live streaming ingest typically incurs a much higher delay (several seconds), and many broadcasting apps buffer frames to a large extent. The use of UDP/WebRTC is also less common for live video ingest (§2.4).

Video on demand shares content distribution paths with live streaming. There have been recent studies on measuring [60, 62], inferring [7, 32, 33, 36], and improving [34, 40, 63] on-demand video streaming QoE. [60] built a general testbed to measure commercial VoD services by leveraging the common features of video players and distribution standards. However, live video ingest has different characteristics and new challenges for measurements. [62] measured ABR rate adaptation on the distribution path. Although [62] supports streaming real-time contents, its "ingest path" is simplified to receiving live TV signals over antennas.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we explore the first-mile ingest aspect of live streaming. We develop Livelyzer, a generalized active measurement and black-box testing framework for analyzing the performance of this component in popular live streaming software and services under controlled settings. We use Livelyzer to characterize the ingest behavior and performance of several live streaming platforms, identify design deficiencies that lead to poor performance, and propose best practice design recommendations to improve the same.

Future directions include evaluating the live ingest pipeline for other types of different network conditions beyond LTE, *e.g.,* 5G, broadband, *etc.* Another direction would be conducting in-depth exploration of the different streaming protocols used on the ingest path, *e.g.,* DASH-IF Live Media Ingest.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and Peshala Pahalawatta for their valuable comments. This material was based upon work partially supported by NSF under grants CCF-1628991 and CNS-1629763.

REFERENCES

[1] ITU-T P. 910. 2008. Subjective Video Quality Assessment Methods for Multimedia Applications.

[2] Adeel Abbas. 2020. Introducing VMAF percentiles for video quality measurements. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/introducing-vmaf-percentiles-for-video-quality-measurements.html

[3] Adobe. 2012. Adobe's Real Time Messaging Protocol. https://www.adobe.com/content/dam/acom/en/devnet/rtmp/pdf/rtmp_specification_1.0.pdf.

[4] Roman Arutyunyan. 2020. NGINX-based Media Streaming Server. https://github.com/arut/nginx-rtmp-module.

[5] Abdelhak Bentaleb, Christian Timmerer, Ali C Begen, and Roger Zimmermann. 2019. Bandwidth prediction in low-latency chunked streaming. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 7–13.

[6] Bitmovin. 2019. Bitmovin Video Developer Report 2019. https://cdn2.hubspot.net/hubfs/3411032/Bitmovin%20Magazine/Video%20Developer%20Report%202019/bitmovin-video-developer-report-2019.pdf.

[7] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2020. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *ACM SIGMETRICS Performance Evaluation Review* 48, 1 (2020), 27–28.

[8] Jan De Cock, Aditya Mavlankar, Anush Moorthy, and Anne Aaron. 2016. A large-scale video codec comparison of x264, x265 and libvpx for practical VOD applications. In *Applications of Digital Image Processing XXXIX*, Vol. 9971. International Society for Optics and Photonics, 997116.

[9] Jie Deng, Gareth Tyson, Felix Cuadrado, and Steve Uhlig. 2017. Internet scale user-generated live video streaming: The Twitch case. In *International Conference on Passive and Active Network Measurement*. Springer, 60–71.

[10] Google Developers. 2020. Real-time communication for the web. https://webrtc.org.

[11] Facebook. 2020. FB: How to Go Live on Mobile? https://www.facebook.com/business/help/1884140525218868.

[12] Facebook. 2020. How do I go live from my Facebook Page? https://www.facebook.com/help/1916203341847533.

[13] Facebook. 2020. How do I set up streaming software to work with Facebook? https://www.facebook.com/help/755943624557739.

[14] Facebook. 2020. What are the video format guidelines for live streaming on Facebook? https://www.facebook.com/help/1534561009906955.

[15] FFmpeg. 2021. A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org

[16] International Organization for Standardization. 2014. ISO/IEC 14496-15:2014 Information technology — Coding of audio-visual objects — Part 15: Carriage of network abstraction layer (NAL) unit structured video in ISO base media file format. https://www.iso.org/standard/65216.html

[17] International Organization for Standardization. 2020. ISO/IEC 23000-19:2020 Information technology — Multimedia application format (MPEG-A) — Part 19: Common media application format (CMAF) for segmented media. https://www.iso.org/standard/79106.html.

[18] DASH Industry Forum. 2020. Low-latency Modes for DASH. https://dashif.org/docs/CR-Low-Latency-Live-r8.pdf.

[19] DASH Industry Forum. 2021. DASH-IF Live Media Ingest Protocol Technical Specification, 26 February 2021. https://dashif-documents.azurewebsites.net/Ingest/master/DASH-IF-Ingest.html.

[20] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. 2018. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*.

[21] Maximilian Grüner, Melissa Licciardello, and Ankit Singla. 2020. Reconstructing proprietary video streaming algorithms. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*.

[22] Oliver L Haimson and John C Tang. 2017. What makes live events engaging on Facebook Live, Periscope, and Snapchat. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 48–60.

[23] Dexter Tan Guan Hao. 2019. Mixer's Faster Than Light streaming protocol explained. https://dotesports.com/streaming/news/mixers-faster-than-light-streaming-protocol-explained.

[24] Cisco Visual Networking Index. 2019. Forecast and Methodology 2017–2022. *Cisco: San Jose, CA, USA* (2019).

[25] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. 2020. Neural-Enhanced Live Streaming: Improving Live Video Ingest via Online Learning. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 107–125.

[26] Emily Krings. 2021. What is RTMPS and Why is it Important to Secure Streaming? https://www.dacast.com/blog/rtmps-streaming/#:~:text=that%20it%20provides.-,RTMP%20vs.,stream%20with%20the%20secure%20alternative

[27] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. 2016. Toward A Practical Perceptual Video Quality Metric. https://medium.com/netflix-techblog/toward-a-practical-perceptual-video-quality-metric-653f208b9652

[28] Zhi Li, Christos Bampis, Julie Novak, Anne Aaron, Kyle Swanson, Anush Moorthy, and Jan De Cock. 2018. VMAF: The Journey Continues. https://netflixtechblog.com/vmaf-the-journey-continues-44b51ee9ed12

[29] Joe Yuchieh Lin, Tsung-Jung Liu, Eddy Chi-Hao Wu, and C-C Jay Kuo. 2014. A fusion-based video quality assessment (FVQA) index. In *Signal and Information Processing Association Annual Summit and Conference (APSIPA), 2014 Asia-Pacific*. IEEE, 1–5.

[30] Tsung-Jung Liu, Yu-Chieh Lin, Weisi Lin, and C-C Jay Kuo. 2013. Visual quality assessment: recent developments, coding applications and future trends. *APSIPA Transactions on Signal and Information Processing* 2 (2013).

[31] Xing Liu, Bo Han, Feng Qian, and Matteo Varvello. 2019. LIME: understanding commercial 360° live video streaming services. In *Proceedings of the 10th ACM Multimedia Systems Conference*. 154–164.

[32] Tarun Mangla, Emir Halepovic, Ellen Zegura, and Mostafa Ammar. 2020. Drop the packets: using coarse-grained data to detect video performance issues. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 71–77.

[33] Tarun Mangla, Ellen Zegura, Mostafa Ammar, Emir Halepovic, Kyung-Wook Hwang, Rittwik Jana, and Marco Platania. 2018. VideoNOC: Assessing video QoE for network operators using passive measurements. In *Proceedings of the 9th ACM Multimedia Systems Conference*. 101–112.

[34] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 197–210.

[35] Matthew K Mukerjee, David Naylor, Junchen Jiang, Dongsu Han, Srinivasan Seshan, and Hui Zhang. 2015. Practical, real-time centralized control for cdn-based live video delivery. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 311–324.

[36] Ashkan Nikravesh, Qi Alfred Chen, Scott Haseley, Xiao Zhu, Geoffrey Challen, and Z Morley Mao. 2018. QoE inference and improvement without end-host control. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 43–57.

[37] Yong Niu, Yong Li, Depeng Jin, Li Su, and Athanasios V Vasilakos. 2015. A survey of millimeter wave communications (mmWave) for 5G: opportunities and challenges. *Wireless networks* 21, 8 (2015), 2657–2676.

[38] Roger Pantos. 2020. HTTP Live Streaming 2nd Edition. https://tools.ietf.org/html/draft-pantos-hls-rfc8216bis-07.

[39] Karine Pires and Gwendal Simon. 2015. YouTube live and Twitch: a tour of user-generated live streaming systems. In *Proceedings of the 6th ACM multimedia systems conference*. 225–230.

[40] Yanyuan Qin, Shuai Hao, Krishna R Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. 2018. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*. 366–378.

[41] Devdeep Ray, Jack Kosaian, KV Rashmi, and Srinivasan Seshan. 2019. Vantage: optimizing video upload for time-shifted viewing of social live streams. In *Proceedings of the ACM Special Interest Group on Data Communication*. 380–393.

[42] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. 2018. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. https://tools.ietf.org/id/draft-ietf-tls-dtls13-01.html.

[43] Selenium. 2021. Selenium automates browsers. https://www.selenium.dev

[44] Matti Siekkinen, Teemu Kämäräinen, Leonardo Favario, and Enrico Masala. 2018. Can you see what I see? Quality-of-experience measurements of mobile live video broadcasting. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 14, 2s (2018), 1–23.

[45] Matti Siekkinen, Enrico Masala, and Teemu Kämäräinen. 2016. A first look at quality of mobile live streaming experience: the case of periscope. In *Proceedings of the 2016 Internet Measurement Conference*. 477–483.

[46] Iraj Sodagar. 2011. The mpeg-dash standard for multimedia streaming over the internet. *IEEE multimedia* 18, 4 (2011), 62–67.

[47] Denny Stohr, Tao Li, Stefan Wilk, Silvia Santini, and Wolfgang Effelsberg. 2015. An analysis of the YouNow live streaming platform. In *2015 IEEE 40th local computer networks conference workshops (LCN Workshops)*. IEEE, 673–679.

[48] OBS Studio. 2020. Open Broadcast Software. https://obsproject.com/.

[49] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. 2012. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology* 22, 12 (2012), 1649–1668.

[50] John C Tang, Gina Venolia, and Kori M Inkpen. 2016. Meerkat and periscope: I stream, you stream, apps stream for live streams. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 4770–4780.

[51] Twitch. 2020. Twitch Recommended Software for Broadcasting. https://help.twitch.tv/s/article/recommended-software-for-broadcasting?language=en_US.

[52] umlaeute. 2020. v4l2loopback - a kernel module to create V4L2 loopback devices. https://github.com/umlaeute/v4l2loopback.

[53] VideoLAN. 2021. x265. https://www.videolan.org/developers/x265.html

[54] Bolun Wang, Xinyi Zhang, Gang Wang, Haitao Zheng, and Ben Y Zhao. 2016. Anatomy of a personalized livestreaming system. In *Proceedings of the 2016 Internet Measurement Conference*. 485–498.
[55] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.
[56] Wikipedia. [n.d.]. Reed–Solomon error correction. https://en.wikipedia.org/wiki/Reed%E2%80%93Solomon_error_correction
[57] Wikipedia. 2020. QR code. https://en.wikipedia.org/wiki/QR_code.
[58] Shichang Xu, Eric Petajan, Subhabrata Sen, and Z Morley Mao. 2020. What you see is what you get: measure ABR video streaming QoE via on-device screen recording. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 60–66.
[59] Shichang Xu, Subhabrata Sen, and Z Morley Mao. 2020. CSI: inferring mobile ABR video adaptation behavior under HTTPS and QUIC. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.
[60] Shichang Xu, Subhabrata Sen, Z Morley Mao, and Yunhan Jia. 2017. Dissecting VOD services for cellular: performance, root causes and best practices. In *Proceedings of the 2017 Internet Measurement Conference*. 220–234.
[61] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. 2012. Video telephony for end-consumers: measurement study of Google+, iChat, and Skype. In *Proceedings of the 2012 Internet Measurement Conference*. 371–384.
[62] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: a randomized experiment in video streaming. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 495–511.
[63] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. 2015. A control-theoretic approach for dynamic adaptive video streaming over HTTP. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.
[64] Youtube. 2020. Create a live stream on mobile – Android – Youtube Help. https://support.google.com/youtube/answer/9228390?hl=en.
[65] Youtube. 2020. Create a live stream via webcam – Youtube Help. https://support.google.com/youtube/answer/9228389?hl=en&ref_topic=9257984.
[66] Youtube. 2020. How to Live Stream On Youtube – How Youtube Works. https://www.youtube.com/howyoutubeworks/product-features/live/#youtube-live.
[67] Youtube. 2020. YouTube Live verified encoders. https://support.google.com/youtube/answer/2907883?hl=en&ref_topic=9257984#zippy=%2Csoftware-encoders.
[68] Cong Zhang and Jiangchuan Liu. 2015. On crowdsourced interactive live streaming: a twitch. tv-based measurement study. In *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*.
[69] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. 2020. OnRL: improving mobile video telephony via online reinforcement learning. In *The 26th Annual International Conference on Mobile Computing and Networking*. 1–14.
[70] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. 2019. Learning to Coordinate Video Codec with Transport Protocol for Mobile Video Telephony. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.