Dashlet: Taming Swipe Uncertainty for Robust Short Video Streaming

Zhuqi Li, Yaxiong Xie, Ravi Netravali, and Kyle Jamieson, *Princeton University*

https://www.usenix.org/conference/nsdi23/presentation/li-zhuqi

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Dashlet: Taming Swipe Uncertainty for Robust Short Video Streaming

Zhuqi Li, Yaxiong Xie, Ravi Netravali, Kyle Jamieson
Princeton University

Abstract

Short video streaming applications have recently gained substantial traction, but the non-linear video presentation they afford swiping users fundamentally changes the problem of maximizing user quality of experience in the face of the vagaries of network throughput and user swipe timing. This paper describes the design and implementation of Dashlet, a system tailored for high quality of experience in short video streaming applications. With the insights we glean from an in-the-wild TikTok performance study and a user study focused on swipe patterns, Dashlet proposes a novel out-of-order video chunk pre-buffering mechanism that leverages a simple, *non* machine learning-based model of users' swipe statistics to determine the pre-buffering order and bitrate. The net result is a system that outperforms TikTok by 28-101%, while also reducing by 30% the number of bytes wasted on downloaded video that is never watched.

1 Introduction

Short video streaming applications like TikTok and YouTube Shorts have rapidly risen in popularity, attracting billions of active users per month [31, 32, 41] and consistently topping popularity lists for mobile apps [33]. Unlike typical video streaming, the median duration of short videos is around 14 seconds [4]. During operation, these apps generate an ordered playlist of short videos (*e.g.*, based on a search or user-specific recommendations), and users watch them serially, with the ability to swipe from one to the next at any time. To provide an immersive experience and keep users engaged, short video streaming applications should minimize the video rebuffering time and maximize the video bitrate, which is modeled by quality-of-experience (QoE) [1, 3, 11, 13].

Although the aforementioned goals are consistent with those in traditional video streaming scenarios, existing ABR algorithms [2, 16, 22, 36, 40] are ill-suited for interactive, short videos. The reason is that predicting user swipes is difficult, and swipe times dictate both which video content will be viewed and when during a session. However, existing algorithms assume that the user will watch content sequentially to completion, and will hence buffer chunks (i.e., multi-second blocks of video) in that order. The deleterious effects, shown in Fig. 1, are twofold: **(1)** many chunks may be downloaded in the current video but never viewed if the user swipes before their playback, wasting resources and adding delays for the chunks that are required, and **(2)** users may swipe to the next video and incur significant rebuffering because that video's chunks have not been downloaded yet.

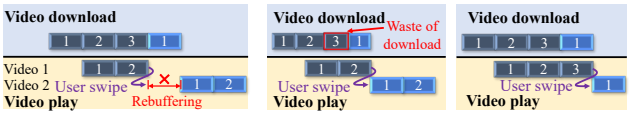


Figure 1: In short video apps, user swipes dictate the playing order of video chunks (and thus, the optimal chunk downloading order).

The fundamental challenge is that there are far too many possible chunk viewing sequences—the user may swipe at any position in each short video, and expects seamless (*i.e.*, no stalls) playback for both the current video, and the next one upon a swipe. The problem thus becomes how to find (at any time during playback) a buffering sequence of chunks in this large search space that maximizes QoE by simultaneously minimizing rebuffering time and wasted bandwidth.

To understand how commercial short streaming platforms attempt to address these challenges, we have conducted a detailed examination of TikTok in the wild (§2). Our key finding is that TikTok does download chunks out of order, but follows a generic algorithm that hedges against immediate rebuffering in the face of fast user swipes (it always pre-buffers the first chunk for the next five videos regardless of network conditions, user patterns, and/or video). This, however, entails substantial QoE penalties and wasted data consumption, as we will show via results from our own study of user swipe patterns across two distinct sets of users on a college campus and Amazon Mechanical Turk (§3). Specifically, we find substantial heterogeneity in the swipe patterns across users, with each warranting a different chunk downloading strategy.

A naïve solution would be to simply predict user swipes—if accurate, this would reduce the problem to a traditional streaming setting since chunk viewing sequences would be known a priori. However, predicting user behavior in interactive applications has consistently proven to be difficult [6, 21, 26]. Instead, we take a more fundamental approach that is rooted in an understanding of where swipe predictions are actually helpful (and actionable).

We present **Dashlet**, a new video streaming algorithm for short video applications (§4). The underlying insight behind Dashlet is that application playback constraints predetermine the relative priorities between many chunks that are candidates for buffering. More specifically, **(1)** later chunks in a video are only reachable via earlier ones, and **(2)** later videos are only reachable via swipes from earlier ones. To prioritize among the remaining chunks, *e.g.*, the next chunk in a given video *vs.* the first chunk in the next video, only coarse grained

information about swipe timings in videos is required. We show, via our user study, that although users tend to exhibit multimodal swipe patterns (complicating chunk prioritization) across videos, distributions from aggregating users' swipes *per video* provide a clear enough signal about which mode to expect. This information is readily available to current short video platforms, and our finding is spiritually aligned with past studies that highlight similarities in user engagement for certain video content [35, 43].

Building on this, Dashlet develops functions that characterize the expected rebuffering time for each potential chunk that could be downloaded, as a continuous function over both the expected download and playback times. These functions embed the aforementioned inter-chunk relationships, as well as rough swipe likelihoods at video start and end. Using these functions, Dashlet employs a greedy algorithm to determine the set of ordered chunks that should be downloaded in the current time horizon to minimize expected rebuffering delays for a given network estimate and across potential viewing sequences. This buffer sequence then feeds directly into a traditional ABR algorithm, which determine bitrates for those chunks that maximize overall QoE. Dashlet further improves upon existing short video systems by not prematurely binding bit rate decisions across entire short videos, and not letting the network idle at any point in time.

We have implemented Dashlet in the DASH framework [8], and compare with the TikTok mobile app with both a human subjects study and a trace-drive study[1]. Across these conditions, we find that Dashlet outperforms TikTok by 28-101% in QoE values, including 8-39% improvement on video bitrate, 1.6-8.9× reduction on rebuffering penalty, and 30% reduction on data wastage. Dashlet's QoE improvement varies with the network throughput, *i.e.*, 543.7%, 221.4%, and 36.6% over TikTok when the throughput is 2-4, 4-6, and 10-12 Mbps, respectively. The improvement diminishes with throughput approaching to 20 Mbps because both Dashlet and TikTok are getting closer to optimum. Further, Dashlet is tolerant to errors in swipe distributions: QoE degradations are only 10% with distribution errors of 50%. We will open source our datasets and implementation post publication.

2 A TikTok Case Study

We examine how TikTok, a state-of-the-art short video app, operates. We first describe its basic architecture (§2.1), before analyzing its operation and limitations (§2.2).

2.1 Short Video Streaming Primer

Unlike traditional streaming apps that divide video into chunks of equal time duration, TikTok splits each video into size-based chunks. For each supported bitrate, if the video file is smaller than 1 MB, TikTok treats the entire video as one chunk; else, the first chunk is the first MB, and the remaining video becomes the second. This chunking strategy enhances reliability, as TikTok pre-buffers first chunks (to cope with swipe uncertainty) so chunking in terms of bytes eliminates first-chunk size variance from variable bitrate encoding.

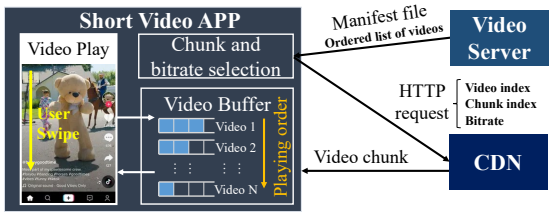


Figure 2: System architecture of TikTok and other short video apps.

Upon receiving a client session request either via a keyword search or category selection (*e.g.*, recommended videos), the server generates an ordered list of short videos to serve (Fig. 2). The server then ships a *manifest file* to the client which embeds the URL, as well as information about the number of *chunks* (multi-second blocks of video) and available bitrates, per video in the ordered list. The client operates much like a traditional streaming player (*e.g.*, DASH), maintaining a playback buffer for downloaded video and employing an adaptive bitrate (ABR) algorithm to determine what chunk to download next, when, and at what bitrate.

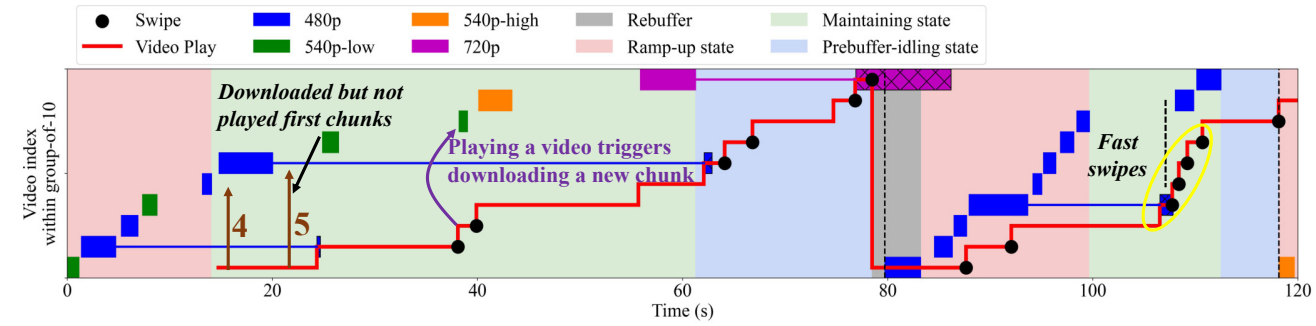
A key difference between traditional and short video streaming is that the client maintains one logical buffer per video in the server-provided manifest file, which contains information for an ordered group of 10 videos. The client requests a new manifest file after it downloads all the first chunks of the videos in the current manifest. Video playback operates sequentially within each logical buffer and across buffers (in the specified order); user swipes and video completion trigger the playback to move to the head of the buffer for the next video. To cope with such semantics, ABR algorithms for short videos have the ability to download chunks for any of the videos in the manifest file at any time.

TikTok provides four bitrate options for each video: 480p, 560p low, 560p high, and 720p, with bitrate adaptation occurring only at video-level (and not chunk-level) granularity. We hypothesize this is because the first 1 MB of a video encoded at different bitrates corresponds to different time durations, precluding seamless bitrate switches for the latter chunk, *i.e.*, content would be missed or repeated. As we will discuss, such constraints significantly limit TikTok from adapting to variations in network capacity during user sessions.

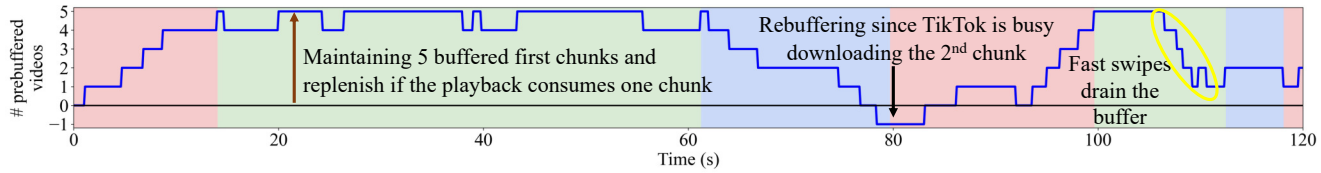
2.2 Analysis of TikTok

To study TikTok in a controlled and systematic manner, we perform our analysis over emulated networks using

[1] We release the code with the following url: https://github.com/PrincetonUniversity/Dashlet under the MIT Open Source License.



(a) Video chunk downloading and playing timeline: video index within a group-of-10 versus wall clock time. The *left and right edge* of the rectangular boxes respectively represent the downloading start and completion times of a chunk, while thin horizontal lines connect first and second chunks (if a second chunk exists), and box *color* indicates bitrate. The solid red line plots video playback.



(b) Client-side buffer occupancy as a function of time, the gap between playback and highest chunk downloaded in **(a)**.

Figure 3: An illustrative video downloading and video playing trace of TikTok, with associated video bitrate and buffer occupancy statistics.

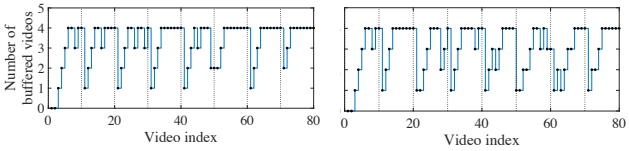
Mahimahi [23]. We log in to TikTok with a two-year old account and mirror its screen to a Linux desktop with scrcpy [28] and use the pyautogui tool [24] to replay aggregated user swipe traces that were collected from our user study (described in §3). During experiments, we use the mitmproxy [5] to collect and decrypt TikTok's network traffic. From the deciphered HTTP messages and headers, we are able to extract for each requested chunk, the video that it pertains to, its index in that video, the requested bitrate, and the download start/end time. Finally, we develop a screen analysis tool using pyautogui and opencv [17] to record duration of each rebuffering event (§5.1 further details our setup).

2.2.1 Chunk Download Control

TikTok's download control algorithm depends both on instantaneous network throughput and the client's internal buffer status: Fig. 3a illustrates its decisions (*i.e.*, order and timing of chunk downloads across videos, bitrates used each time) during a representative two-minute session. We plot client-side playback buffer occupancy in Fig. 3b, which shows the number of videos with at least one downloaded (but unplayed) chunk. We see that TikTok spends most of its time downloading the first chunk of videos, and downloads the second chunk when and only when the video starts to play, *e.g.*, the download of the second chunk of video two and the play-start of video two start simultaneously at $t = 22$ s.

Our analysis indicates that TikTok proceeds according to three discrete states, cycling among the three in order to handle one *group-of-ten* videos. At startup and the start of every group-of-ten, the **ramping-up state** continuously downloads first chunks to build up buffers. After accumulating five first chunks at $t = 18$ seconds, TikTok starts to play the buffered video and enters the **maintaining state**, where it aims to maintain a constant five buffered first chunks. Upon playing a new video (due to user swipe or reaching the end of a video), the client fetches one first chunk from the buffer, triggering TikTok to immediately initiate download of the first chunk of the next video in the manifest, as indicated by the additional download events corresponding to either swipes or video changes due to end of video in the green "maintaining state" regions of Fig. 3a. We see in Fig. 3b that as the downloading of each first chunk finishes, buffer levels return to five, the high water mark buffering level TikTok has chosen. The advantage of the maintaining state is resilience to quick user swipes: in the second group-of-ten of Fig. 3 ($t = 110$), the user swipes early in multiple consecutive videos, quickly draining the buffer, but TikTok experiences no rebuffering since its buffer contains the five first chunks.

Finally, after downloading all the first chunks of the 10 videos listed in the current manifest file, TikTok enters the **prebuffer-idling state**, where it stops initiating any new downloads of first chunks. Meanwhile, TikTok continues video playback, consuming video chunks in its buffer, so buffer occupancy decreases monotonically in this state, as shown in Fig. 3b. Our hypothetical explanation of this idle period is that TikTok is waiting to measure the user's reaction (swiping early means they might not be interested in the content) to the videos TikTok recommends in last round



(a) Net. throughput 10 Mbit/s. **(b)** Net. throughput 3 Mbit/s.

Figure 4: The number of downloaded videos inside the buffer when TikTok starts to download the first chunk of a new video via networks with capacity of **(a)** 10 Mbit/s and **(b)** 3 Mbit/s.

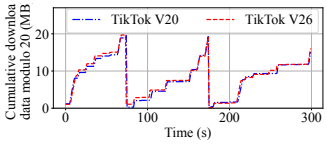


Figure 5: Cumulative downloaded data modulo by 20 MB for TikTok v20.9.1 and TikTok v26.3.3 when playing the same video sequence at the same swipe pace.

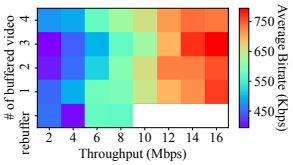


Figure 6: Impact of network throughput and client video buffer occupancy on TikTok's chosen video bitrate.

(manifest file), so it can assess its recommendation quality and adjust the subsequent round's recommendation before sending the next manifest file.

In contrast to the resilience of the maintaining state, TikTok becomes somewhat vulnerable in the prebuffer-idling state, where TikTok drains the buffer by itself. For example, TikTok experiences rebuffering in the middle of two video groups in Fig. 3. At that moment, TikTok has no buffered first chunk and at the same time spends a long time downloading the second chunk of the current video, leaving no time budget for downloading the first chunk of next video. In such a case, one user swipe results in rebuffering.

When the user starts to watch the ninth of the group-of-ten videos listed in a manifest, TikTok exits prebuffer-idle and begins afresh in the ramp-up state to download the videos listed in the next manifest file. The cycle through these three states repeats for each group-of-ten.

2.2.2 Network and Swipe Input Adaptation

We now investigate the effects of swipes, buffer occupancy, and the network on TikTok's bitrate and buffering choices. To measure the impact of the network on buffering strategy, we control network capacity to 10 and 3 Mbit/s using Mahimahi and plot the number of buffered first chunks at the moment TikTok initiates a download of the first chunks, in Fig. 4. Combing Fig. 4a and 4b, we see that TikTok adopts the same buffering strategy regardless of network capacity.

Next, we analyze the joint impact of network throughput and buffering status on Tiktok's bitrate decisions. We collect instantaneous network throughput and buffer status coupled with TikTok's bitrate decisions, for 5,300 videos, and plot the results in Fig. 6. In the figure, the x-axis is the network throughput of the one-second period before the downloading of that video, *i.e.*, the time period within which TikTok makes its decisions about the bitrate. The y-axis is the number of downloaded first chunks in the buffer. The color of a tile represents the average bitrate R of the video, which is given by $R = S/L$ where S is the size of the video in bits and L is the length of the video in seconds. Some tiles are not colored because the combination of the throughput and buffer status is not seen during our measurement, *e.g.*, when the throughput is 16 Mbit/s, we always observe four downloaded first chunks in the buffer. We observe that bitrate decisions correlate positively with network throughput, but observe no evidence for correlation with buffer status.

2.2.3 Buffering logic on different versions of TikTok

Our reverse engineering tool can only decipher complete TikTok telemetry information up to version v20.9.1. To investigate whether there are any updates in the buffering algorithm between v20.9.1 and the newest TikTok version (v26.3.3), we use scripts to watch the same videos on different versions, under the same network throughput and swipe pace. We record the number of bytes downloaded versus time with tcpdump. Fig. 5 shows an example trace for the two versions of TikTok. By correlating the downloading traces at different throughput and swipe speed, we infer that v20.9.1 and v26.3.3 use similar or identical buffering logic. In the rest of the paper, we only present v20.9.1 results.

2.2.4 Limitations of Current Short Video Streaming

Despite pre-buffering the beginnings of short videos, TikTok has a fundamentally static approach to coping with swipe uncertainty, with no evidence for adaptation across different videos or users. This approach is often too cautious or aggressive, manifesting in two particular ways:

Lack of swipe prediction. TikTok prioritizes the downloading of the first chunk, assuming that the user always swipes frequently, and delays the downloading of the second to the beginning of video playback. As we will show next however, there are indeed some users who swipe early when watching a video, but there also a significant number of users who watch most of many videos and swipe at the end or not at all. So, the urgency of downloading the second chunk varies with users and videos: a fixed rule cannot handle all cases.

Premature bitrate binding. TikTok groups the first MB of video data into the first chunk but selects the bitrate for both chunks according to the network conditions present during the first, prematurely binding the system into that bitrate for both. By design, there is often a large time lag between the downloading of the first and second chunks, as discussed above (the median gap between first and second chunk downloads is 25 s., with an interquartile range of 23 s.), resulting in a potential mismatch with network conditions

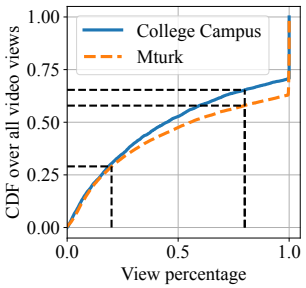


Figure 7: The distribution of average viewing percentage across all short video views.

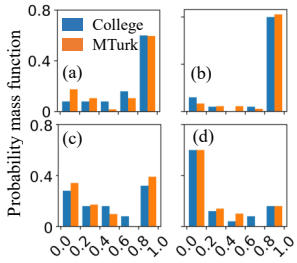


Figure 8: Distribution (across different users) of video viewing percentage for four sample videos *(a)–(d)*.

that change in the meantime.

Network Idling. As shown in Fig. 3a, TikTok has a prebuffer idling state. In the contrast, the buffer is not full and the bitrate of videos still has room to improve. This also calls for a better ABR algorithm to stream higher bitrate video by utilizing the idle time in a better way.

To understand the mismatch between TikTok's generic rule and the varying user swipe patterns, in the next section, we characterize the swipe patterns across real users and videos via two user studies.

3 Characterizing User Swipes

To better understand the nature of user interactions (*i.e.*, swipes) with short video applications, we conducted two IRB-approved user studies. In each study, we present users with a web-based short video streaming service that resembles the interface offered by TikTok We considered 500 popular short videos gathered by crawling the videos displayed on the TikTok landing page over time. The videos were randomly ordered per session, and each user watches 20 minutes of video with the ability to swipe freely (all swipes are recorded). Note that the number of videos watched by a given user depends on the number of swipes they performed.

For generality, we performed two versions of this study:

1. College campus: we recruit 25 student volunteers who collectively swipe 3,069 times during the study.

2. Amazon Mechanical Turk ("MTurk"): we recruit 258 different users. To ensure active user participation, we augment our web application to inject random interactivity tests that ask users to swipe within 10 seconds. Users who fail to swipe in time are entirely excluded from the study; users who do swipe continue the experience, but we exclude the forced swipe(s) from our final dataset. In total, we retain data from 133 workers, which covers 15,344 swipes.

Overall swipe distributions. Fig. 7 shows the distribution of swipe times across all video-user pairs in both studies. Users are most likely to swipe either soon after video playback begins or at the end of the video (manually or via auto-swiping once the video completes); this is consistent with prior studies on user swipe patterns [44]. For instance, 29% and 42% of swipes from MTurk users are within the first 20% or last 20% of videos, respectively. Swipes between these two endpoints occur, but far less often and with increasingly low likelihood as users watch more videos, *e.g.*, only 6% of swipes in the College Campus dataset are in the 60–80% of videos.

Swipe distributions per video. Fig. 8 shows swipe probabilities for four representative videos, aggregated across users who watched each one in the two studies. Different videos yield significantly different swipe distributions: over 60% of swipes in videos (a) and 80% of swipes in videos (d) come within the last few seconds (indicating low swipe probabilities for these). Video (c) exhibits the opposite pattern—60% of swipes in the first 20% of the video (indicating high swipe probabilities)—while swipes in (b) are more evenly distributed in time. Perhaps more importantly, we observe substantial stability in the swipe distributions per video across different user datasets: KL divergence values between the MTurk and College Campus datasets are 0.2 and 0.8 for the median and 95th percentile videos, respectively.

Conclusions. Despite general similarities in swipe patterns, users follow a few different modes of swiping (*e.g.*, swiping early in the chunk *vs.* not at all), each of which warrants a different buffering strategy to ensure high QoE. Fortunately, cross-user swipe data that is aggregated *per video* provides a relatively stable indicator as to how likely swipes are (and will be) in a given video, and (more coarsely) at what part of the video they will occur. We show in §4 how Dashlet leverages this coarse information – which is readily available at existing short video servers – to make robust buffering decisions that handle cross-user swipe traces.

4 Design

Dashlet leverages swipe distribution stability across videos (§3) to get a *coarse* sense of the likelihood of swipes at different video chunks. Coupling this information with constraints on inter-chunk viewing sequences intrinsic to short video applications, Dashlet models the *expected rebuffering time* for each potential chunk as a continuous function over the expected download and playback times (§4.1), then employs a greedy algorithm atop those functions to find a chunk buffering sequence that minimizes expected rebuffering delay over a time horizon for a given network throughput estimate, and across different user viewing sequences. Lastly, Dashlet feeds that buffering sequence into a bitrate selection algorithm (RobustMPC [40] in our implementation) to control video chunk bitrate and optimize overall QoE (§4.2).

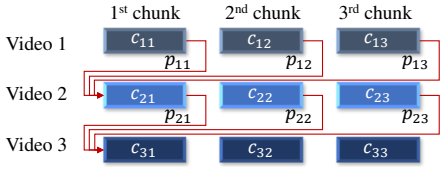


Figure 9: Short video streaming model: the player plays videos sequentially, switching to the first chunk of the next video after a swipe.

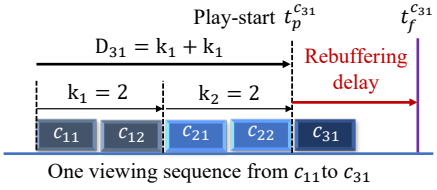


Figure 10: Chunk rebuffering delay depends on the order between the play start time t_p and the download finish time t_f.

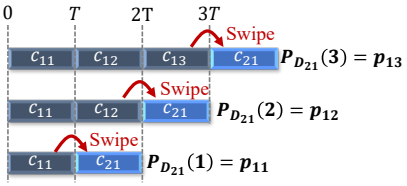


Figure 11: Three possible viewing sequences that start from chunk c_{11} and end at chunk c_{21}.

4.1 Forecasting Rebuffering Delay

Dashlet's expected rebuffering functions aim to quantify user-perceived delays across different chunk download times and viewing sequences. We begin by explaining the construction of these functions in a discrete setting where users can only swipe at chunk boundaries; we then extend the discussion to incorporate arbitrarily-timed user swipes.

System Model. Short video apps follow the flow shown in Fig. 9. Each video consists of multiple chunks of *chunk time* T. Within the i^{th} video with N_i chunks, if the user does not swipe, the video player plays its chunks c_{ij} sequentially, where $j \in [0, N_i]$ is the chunk index. When playback reaches the end of the video or the user swipes, the player jumps to the first chunk of the next video. Since user swipe distributions vary across videos (§3), we denote the probability that the user swipes after watching chunk c_{ij} as p_{ij}. The list of the chunks the user watches is a *viewing sequence*

$$V_s = [c_{11}, \ldots, c_{1k_i}, c_{21}, \ldots, c_{K1}, \ldots, c_{Kk_L}] \tag{1}$$

where the user views the first k_i chunks of the i^{th} video, assuming that the user watches L videos in total. Then the probability distribution of k_i is $P_{k_i} = \{p_{i1}, p_{i2}, \ldots, p_{iN_i}\}$. We define D_{ij}, the number of chunks that a user has watched prior to chunk c_{ij}:

$$D_{ij} = \sum_{l}^{i-1} k_l + (j-1). \tag{2}$$

By knowing the number of chunks that a user has watched before c_{ij}, the playback start time of c_{ij} then will be $D_{ij} \cdot T$. As shown in Fig. 10, the expected rebuffering delay for some chunk c depends on the relationship between the chunk's *play start* time t_p^c and *download finish* time t_f^c. There exists no rebuffering if the chunk downloading finishes before the play start time. Otherwise, rebuffering happens and the time difference between t_p^c and t_f^c tells us c's *rebuffering delay*:

$$T_c^{\text{rebuf}}(t_f^c, t_p^c) = \begin{cases} 0, & t_f^c < t_p^c \\ t_f^c - t_p^c, & t_f^c \geq t_p^c \end{cases} \tag{3}$$

The play start time of each chunk is determined by the viewing sequence V_s, as shown in Fig. 10. Since our goal is to schedule c's download to minimize rebuffering, we now formulate a reward function to meet this goal, parameterized on t_f^c and averaging over all possible viewing sequences (which are not under our control). The *expected rebuffering delay* of chunk c given that chunk's download finish time t_f^c, across all possible viewing sequences, is:

$$\mathbf{E}_c^{\text{rebuf}}(t_f^c) = \sum_{V_s \in \Phi} \Pr(V_s) \cdot T_c^{\text{rebuf}}(t_f^c, t_p^c(V_s)) \tag{4}$$

where probability $\Pr(V_s)$ represents how likely a specific viewing sequence V_s will appear based on user swipe distribution data, $t_p^c(V_s)$ is c's play start time in V_s, and Φ is the set of all possible viewing sequences.

To calculate the expected rebuffering delay for a specific chunk, we enumerate all possible viewing sequences that reach this chunk, as Eq. 4 shows. For each sequence, we compute how likely this sequence will appear based on user swipe distributions, and then determine the play start time of that specific chunk. Based on short video chunk playback constraints (§1, p.), we propose separate algorithms for calculating the expected rebuffering delay of a video's first chunk, and remaining chunks, respectively.

First chunk of a video. The number of possible viewing sequences between chunk one of video one (c_{11}) and chunk one of video i (c_{i1}) increases exponentially with i. On the other hand, the number of sequences from the first chunk of the previous to the first chunk of the current video is bounded by the number of chunks in the former. For example (see Fig. 9), there are three possible viewing sequences from chunk c_{21} to c_{31}. We therefore enumerate the viewing sequences in a recursive manner: deriving the viewing sequences that reach the first chunk of the i^{th} video based on the viewing sequence of the first chunk of the $(i-1)^{\text{st}}$ video.

We start from the base case, viewing sequences from c_{11} to c_{21}. Fig. 11 lists all three possible viewing sequences that start from c_{11}: we see that random variable $D_{21} = k_1$ (*cf.* Eq. 2). Similarly, as shown in Fig. 10, $D_{31} = D_{21} + k_2$ (*cf.* Eq. 1). The distribution of D_{31} is then:

$$P_{D_{31}}[n_0] = \sum_{i=1}^{n_0-1} P_{D_{21}}[i] \cdot P_{k_2}[n_0 - i] \tag{5}$$

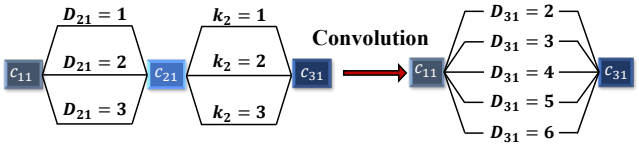


Figure 12: Convolution of the $P_{D_{21}}[\cdot]$ and $P_{K_2}[\cdot]$ provides us the probability distribution $P_{D_{31}}[\cdot]$.

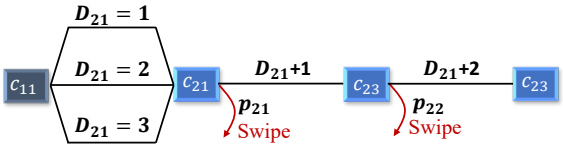


Figure 13: Starting from chunk c_{21}, the user must watch the second video continuously with swiping to reach chunk c_{23}.

where $P_{D_{31}}[n_0]$ means the probability of there are n_0 chunks before chunk c_{31} is viewed. This formula by definition is the operation of convolution between D_{21} and k_2, as shown in Fig. 12. Without losing generality, the number of chunks that user watches before chunk c_{i1}, D_{i1}, is $D_{(i-1)1}+k_{i-1}$. Therefore the distribution of D_{i1} is:

$$P_{D_{i1}} = P_{D_{(i-1)1}} * P_{k_{i-1}}. \tag{6}$$

With the knowledge of D_{i1}'s distribution for the first chunk of all the videos, we calculate the expected rebuffering delay of chunk c_{i1}, as the function of download finish time:

$$\mathbf{E}^{\text{rebuf}}_{c_{i1}}(t_f) = \sum P_{D_{ij}}[n] \cdot T^{\text{rebuf}}_{c_{i1}}(t_f, (n+1)T) \tag{7}$$

Remaining chunks in a video. There exists only one viewing sequence from the first to later chunks of the same video: Fig. 13 shows that c_{23} will be played when and only when the user watches the $i=2^{\text{nd}}$ video continuously without swiping. For non-first chunk c_{ij}, the number of chunks that user watched before it, D_{ij}, is the summation of D_{i1} and $j-1$ since the user has to watch the first $j-1$ chunks in video i before starting to watch it. Then the distribution of D_{ij} is that of D_{i1}, delayed by $j-1$ chunks. In addition, the user might swipe to the next video before watching c_{ij}:

$$P_{D_{ij}}[n_0] = P_{D_{i1}}[n_0-(j-1)] \times (1-\sum_{m=1}^{j-1} p_{im}). \tag{8}$$

With the distribution of D_{ij}, we follow the same procedure to calculate expected rebuffering time for remaining chunks in a video, according to Eq. 7.

Arbitrary user swipes. In reality, swipes do not only happen after a chunk finishes. If the continuously-valued viewing time for video i is κ_i, the PDF of κ_i is $f_{\kappa_i}(t_0)$. The *play start time* of c_{ij}, Δ_{ij}, is a random variable, with PDF $f_{\Delta_{ij}}(t)$. For the first chunk of video i, its playing start time $t_f^{c_{i1}}$ is also the summation of the playing start time of the previous video $t_f^{c_{(i-1)1}}$ and the time the user spends watching the previous video κ_{i-1}. Following a similar principle, we compute $f_{\Delta_{i1}}(t)$ for the first chunk of a video i as

$$f_{\Delta_{i1}}(t) = f_{\Delta_{(i-1)1}}(t) * f_{\kappa_{i-1}}(t). \tag{9}$$

For subsequent chunks c_{ij}, we also calculate the playing start distribution based on the first chunk

$$f_{\Delta_{ij}}(t) = f_{\Delta_{(i-1)i}}(t-(j-1)\cdot L)\cdot\left(1-\int_0^{(j-1)\cdot L} f_{\kappa_i}(x)\right)dx \tag{10}$$

Then the expected rebuffering function can be calculated similarly to Eq. 7:

$$\mathbf{E}^{rebf}_{c_{ij}}(x) = \int_{t=0}^{x} f_{\Delta_{ij}}(t) \times T^{rebf}_{c_{ij}}(x,t)dt \tag{11}$$

In the implementation, we approximate the continuous value swipe distribution with a discrete distribution with the time granularity of 0.1 seconds. The integral then can be approximated by the summation in the discrete distribution.

4.2 Determining Buffering Sequences

Given the preceding computation of expected rebuffering delay for each chunk, Dashlet's next task is to determine an order of chunks to download (*i.e.*, a buffering sequence) that minimizes expected rebuffering delay over a lookahead horizon. Prior schemes (*e.g.*, MPC [40]) can then be used to determine the bitrates for those chunks to optimize overall QoE for the horizon. However, unlike prior schemes, the horizon that we use is based on time (not chunks), since different user swipe patterns can translate into different numbers of viewed chunks. Using a horizon sized to a fixed number of chunks could result in optimization over very short viewing times (negating the effects of longer-term planning). Our current implementation uses a lookahead window of 25 seconds based on empirical observations, which is equivalent to the five chunks MPC uses. Chunk ordering relies primarily on whether the user swipes near the beginning of the video or not: *e.g.* if the user is highly likely to not swipe in c_{11}, the algorithm then needs to prioritize c_{12} over c_{21}.

4.2.1 Selecting the candidate chunk set

To determine the set of chunks to consider, we enforce a threshold on the minimum rebuffering penalty that each chunk is expected to incur at the end of the horizon if it is not included in the buffer sequence (Fig. 14(a)). Chunks whose rebuffering penalty falls below the threshold are deemed as unlikely to be viewed during the horizon (c_{32} in Fig. 14(a)), and thus low priority for inclusion in the buffer sequence.

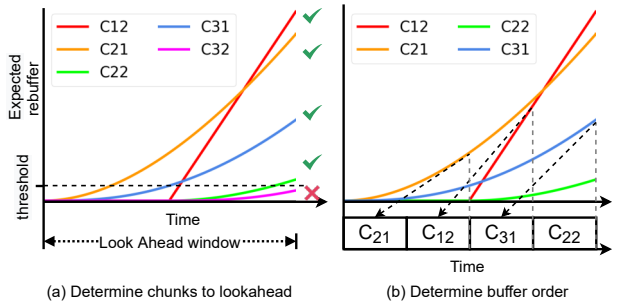


Figure 14: An example to illustrate Dashlet's algorithm.

Note that buffer sequences are constructed each time a chunk download completes, so an excluded chunk for one horizon may still be downloaded shortly (via inclusion in the next horizon's buffer sequence). We use an empirically-configured value of $1/\mu$ for threshold, which is the inverse of the rebuffering penalty weight in our target QoE function.

Using the set of chunks to consider, our final task is to order them in a manner that minimizes expected rebuffering penalties. We assign a bitrate to each chunk, and then use estimated network bandwidth to determine when it will complete donwloading (assuming some start time). This allows us to compute expected rebuffering time per chunk (§4.1). However, to bound computational complexity (since download decisions must be fast) we temporarily assume an equal bitrate per chunk that is set to the maximum bitrate, which ensures that all chunks in the list will complete downloading before the horizon completes. Although exclusion of per-chunk bitrate decisions here can result in suboptimal orderings, these effects are marginal (evidenced by Dashlet's closeness to the Optimal scheme in §5.2), as priorities between chunks (and potential per-chunk viewing times) are largely dictated by viewing constraints imposed by the application (§4.1). Thus, minor discrepancies in bitrates across chunks are unlikely to flip the priority order among them.

4.2.2 Priority-ordering the buffer sequence

To sort our list of chunks into a buffer sequence, we follow a greedy algorithm, whereby we partition the horizon into chunk-sized slots. For a given slot i, we select the chunk that will incur the largest additional rebuffering penalty if it were to be scheduled in slot $i+1$ rather than i. Fig. 14(b) shows this process for a scenario in which chunk c_{11} just completed downloading: c_{21} is assigned to slot 1 as its rebuffering penalty jumps the most between slots 1 and 2; c_{12} is next as it has the highest penalty for not going in slot 2, and so on. Finally, using the generated buffer sequence, Dashlet applies MPC's algorithm to determine the bitrate for each chunk in the buffer sequence in a way that optimizes the entire QoE (not just minimizing rebuffering) for the horizon according to the forecasted network throughput, *i.e.*, the harmonic mean over the observed throughputs in the last 5 chunk downloads. We describe the above algorithm with a pseudo code in §A.

4.3 Implementation

Dashlet's implementation includes one control module and multiple buffer modules. The control module schedules the chunk downloading and the buffer modules reuses the DASH.js playback management implementation to download video chunks. §B provides more implementation details.

5 Evaluation

We evaluate Dashlet across a variety of mobile network conditions, real user swipe traces, and videos. Our key findings:

- Dashlet outperforms TikTok by 28-101% in terms of average QoE, including 8-39% improvement on video bitrate, 1.6-8.9× reduction on rebuffering penalty, and 30% reduction on data wastage.
- Dashlet's QoE improvement varies with the network throughput, *i.e.*, 543.7%, 221.4%, and 36.6% over TikTok when the throughput is 2-4, 4-6, and 10-12 Mbps, respectively. The improvement diminishes with throughput approaching to 20 Mbps because both Dashlet and TikTok are getting closer to optimum.
- Dashlet tolerates errors in swipe distributions: with errors of 50%, Dashlet makes the correct buffering decisions 96.5% of the time, yielding an QoE reductions of only 10%. compared to cases with no distribution errors.

5.1 Methodology

Baselines. We compare Dashlet with the following systems:

- *TikTok.* We compare with TikTok App (version v.20.9.1).
- *Oracle.* We also run an 'oracle' baseline that serves as an upper bound for QoE. The oracle is the RobustMPC algorithm [40] running with perfect (a priori) knowledge of both the user swipe traces and network throughput in each experiment. With that information, the algorithm knows the upcoming video viewing sequence at all times, and can thus pick the buffer sequences (and bitrates) that directly optimize QoE for the current network conditions.

Overall setup. All video clients run on a rooted Pixel 2 phone (Android 10). The Oracle algorithm and Dashlet run in the Google Chrome browser (v. 97.0.4692.87), and contact a local desktop which houses the videos accessed in each experiment (described below). In contrast, TikTok runs as an unmodified, native Android app and contacts Akamai CDNs to fetch video content as it normally does. We checked the location of the CDN content server node and verified it was local to our area. All traffic to and from the phone passes over emulated mobile networks (which run atop WiFi connections with average speeds of ≈300 Mbps); to compensate for the discrepancy in video servers, we added 6 ms of round trip

delay to traffic for Dashlet and the Oracle algorithm, which reflects the maximum ping time we observed to the CDN used by TikTok.

Evaluation metrics. Short videos share similar goals of traditional video streaming [22, 40]: maximizing video bitrate, minimizing rebuffering delays, and avoiding frequent bitrate fluctuations, so we adopt a widely used QoE metric:

$$QoE = R_{bitrate} - \mu \cdot P_{rebuffer} - \eta \cdot P_{smooth} \quad (12)$$

where $R_{bitrate}$ is the average video bitrate, $P_{rebuffer}$ is the cumulative penalty for rebuffering (i.e., stalled playback), and P_{smooth} is the penalty for frequent bitrate switching across adjacent chunks. We use the same values for μ and η as prior work [40], i.e., $\mu = 3000$ and $\eta = 1$.

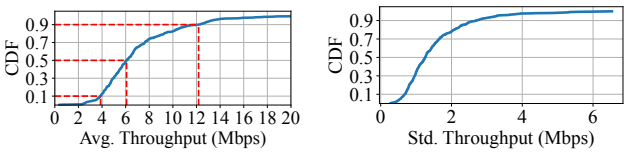
Human subjects study for QoE. We conduct a small-scale human study, where we recruit ten participants[2]. We ask them to log in their own accounts to use TikTok under emulated mobile networks (the videos are recommended by TikTok)[3]. We randomly choose three network traces with average throughput of 4 ± 0.1, 6 ± 0.1, 12 ± 0.1 Mbps respectively. We record the content, quality, order of videos TikTok streams to each user, and swipe timestamps. For the evaluation of Dashlet, we first download every video that users have watched in TikTok experiments and collect per-video user swipe distributions with Amazon Mturk. We then stream the same videos in the same order as TikTok using Dashlet, under the same emulated network, during which we replay the user swipes recorded in the TikTok experiment. Take the analogy to machine learning: the "training set" we use for Dashlet is collected by MTurk, and the testing set is real users' swipe. To quantify performance, we record the quality of every video chunk and the rebuffering event to calculate the QoE for both TikTok and Dashlet.

Human subjects study for users' satisfaction. We let the same group of participants use TikTok and Dashlet for in total 30 minutes and ask them to rate the video quality and smoothness after they finish. Each participant used both TikTok and Dashlet for three five-minute sessions under three different network traces. Notice that the videos played in TikTok and Dashlet are different in the study since the users would behave differently if shown the same video once more, e.g., users tend to swipe fast when they are already familiar with the content in a video. For Dashlet, the swipe distribution is pre-collected with MTurk before the study.

Trace-driven study. We run a trace-driven study to scale up the evaluation under different user swipe speed and network traces. We use a script to automatically swipe in TikTok

[2]Among the ten participants, three of them are new users, three of them are occasional users, and four of them are daily users.

[3]The only action that users perform in the study is to swipe to the next video based on their watching experiences



(a) Average. **(b)** Standard deviation.

Figure 15: Throughput distribution for our network dataset.

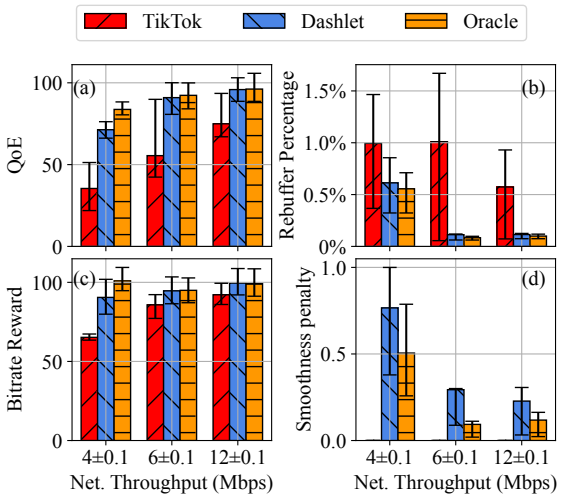


Figure 16: End-to-end result for human subjects study. (a) QoE (b) Rebuffer percentage. (c) Bitrate reward (d) Smoothness penalty

based on the distribution shown in Fig. 7. In order to enforce the same playing sequence (i.e., ordered list) of videos across the considered systems, we exploit the fact that the order in which videos are streamed with TikTok for a given keyword search remains unchanged on the order of many days. We use that same order across all systems and across experiments with different network and swipe traces. Each experiment considers 10 minutes of viewing time to match the average session time for TikTok users [34]. Similar to our human subjects study, we replay the same traces recorded from TikTok experiments to evaluate Dashlet and Oracle. The swipe distribution used for Dashlet in replay is collected from another batch of user study via Amazon Mturk.

Network conditions. We consider the combination of two sets of mobile network traces: (1) the FCC LTE dataset [9] that is widely used in prior work [22, 40], and (2) a WiFi trace dataset that we collected in January 2022 in a shopping mall. Fig. 15 shows the average and standard deviation of throughput traces in the combined dataset.

5.2 End-to-End performance

Human subjects study. Fig. 16 shows the end-to-end result for human subjects study, including QoE, rebuffering

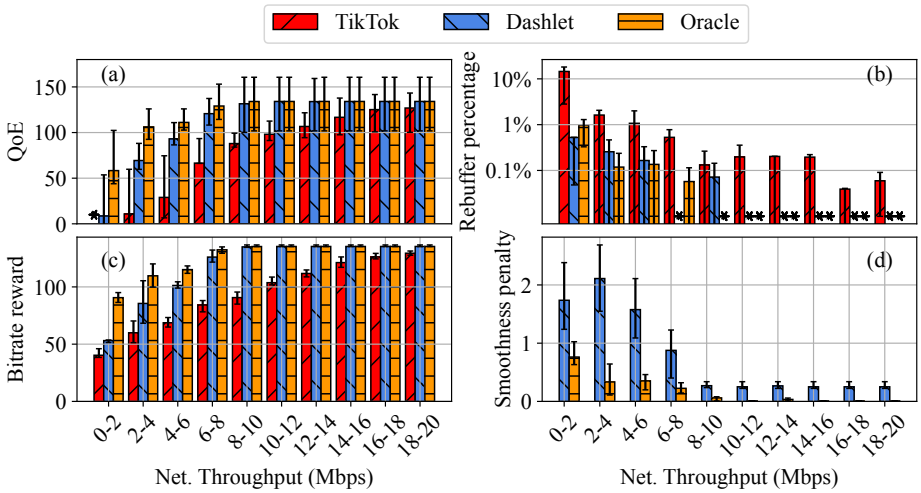


Figure 17: End-to-end result for trace-driven study. (a) QoE. * is the outlier data point with average QoE at -389 due to rebuffer (b) Rebuffer percentage (of total time). Note log ordinate axis; * denotes zero rebuffering. (c) Bitrate reward. (d) Smoothness penalty

Network throughput (Mbps)	**4 ± 0.1**	**6 ± 0.1**	**12± 0.1**
TikTok quality	3.1 ± 0.83	3.2 ± 0.87	4.0 ± 0.89
Dashlet quality	3.6 ± 0.80	3.9 ± 0.70	4.1 ± 0.94
TikTok stall	2.8 ± 1.08	3.0 ± 0.77	4.2 ± 0.99
Dashlet stall	3.5 ± 1.02	3.9 ± 0.94	4.3 ± 0.90

Table 1: User survey for TikTok and Dashlet. Each participant is asked to score 1 (worst) to 5 (best) in terms of video quality (resolution) and stall (rebuffer) under three different network throughput. The sample questionnaire is shown in §D. Table summarizes the average and standard deviation of the score.

Network throughput (Mbps)	**4 ± 0.1**	**6 ± 0.1**	**12± 0.1**
QoE	-363.2	-287.9	-133.5
Rebuffer percentage	28.0%	24.8%	14.3%
Bitrate reward	77.2	96.6	97.8
Smoothness Penalty	0.38	0.12	0.02

Table 2: End-to-end result for MPC.

percentage, bitrate reward and smoothness penalty. There are two key takeaways from these results. First, Dashlet consistently outperforms TikTok across different network throughput. Dashlet improve the average QoE over TikTok by 101%, 64%, 28% on 4 Mbps, 6 Mbps, 12 Mbps respectively. When break down the QoE into the components, Dashlet reduces the rebuffering by 1.6-8.9x compared with TikTok and improve the QoE by 8% - 39% with the cost of marginal smoothness penalty. Second, Dashlet can reach the close-to-optimal performance starting from 6 Mbps. While TikTok does not achieve that even at 12 Mbps.

We also run experiments on MPC [40], a state of art traditional video streaming algorithm, on the same setup mentioned above. As a traditional video streaming algorithm, MPC only prebuffers chunks for the current video. Table 2 summarizes the end-to-end result. Compared with Dashlet, MPC incurs a much higher rebuffering as it experiences rebuffer delay every time the user swipes to a new video., leading a significant lower QoE compared with Dashlet.

We also perform a experiment to understand the participants' satisfaction of the service provided by TikTok and Dashlet. We let the participants watch videos using TikTok and Dashlet for five minutes, after which we conduct a user survey by asking the participant to report their satisfaction scores in terms of video quality (resolution) and stall (rebuffer) conditions for both TikTok and Dashlet. Table 1 shows the users' satisfaction towards the video resolution and rebuffer for both TikTok and Dashlet on the human subjects study. From the figure, we can see that Dashlet improves the users satisfaction on both video resolution and rebuffering.

Trace-driven study. Fig. 17 shows the result for trace-driven study. Key results are: (1) Dashlet's QoE improvement varies with the network throughput, *i.e.*, 543.7%, 221.4%, and 36.6% over TikTok when the throughput is 2-4, 4-6, and 10-12 Mbps, respectively. The improvement diminishes with throughput approaching to 20 Mbps. (2) Dashlet can reach the optimal QoE at a much lower network throughput than TikTok, *i.e.* Dashlet reaches the optimal at throughput 8-10 Mbps. While TikTok is close to the optimal at the throughput 18-20 Mbps. (3) Dashlet consistently incurs a lower rebuffering compared with TikTok.

5.3 Ablation study

We further perform an ablation study to understand the contribution of five design components (detailed in Table 3). *Idle:* TikTok has a prebuffer idle state as described in §2.2.1 while Dashlet does not. *Chunking:* TikTok splits the video into one or two chunks while Dashlet splits the video into

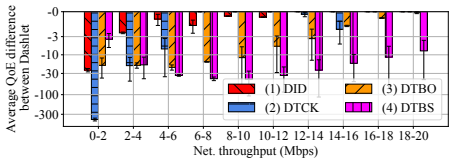


Figure 18: The contribution to End-to-End QoE for different design components. The y-axis in log scale is the QoE difference between the corresponding ablation study systems and Dashlet.

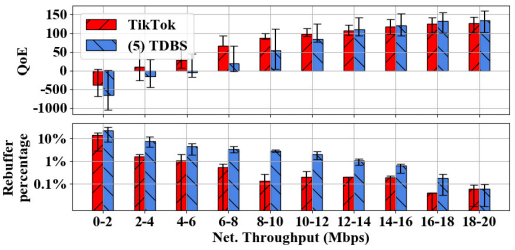


Figure 19: Ablation study for bitrate choice.

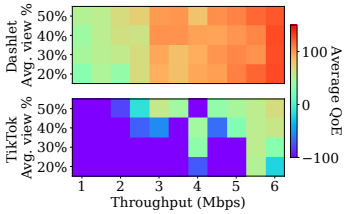


Figure 20: Average QoE of Dashlet and TikTok under different average viewing percentage (based on swipe patterns) and network throughput.

System Name	Idle	Chunking	Fix bitrate	Buffer order	Bitrate selection
(1) Dashlet+Prebuffer idle (DID)	T	D	D	D	D
(2) Dashlet+TikTok Chunking (DTCK)	D	T	T	D	D
(3) Dashlet+TikTok buffer order (DTBO)	D	D	D	T	D
(4) Dashlet+TikTok bitrate (DTBS)	D	D	D	D	T
(5) TikTok+Dashlet bitrate (TDBS)	T	T	T	T	D

Table 3: Setup for ablation study. We summarize the difference in design components between Dashlet and TikTok and evaluate the impact of corresponding design components. **"T"** and **"D"** denote TikTok's and Dashlet's design components respectively.

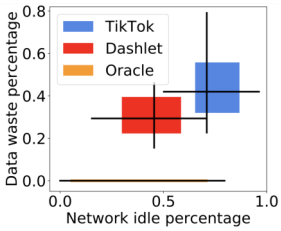


Figure 21: Data wastage and network idle time. Boxes span 25-75th percentiles. Black lines span min/max, and intersect at the median for both properties.

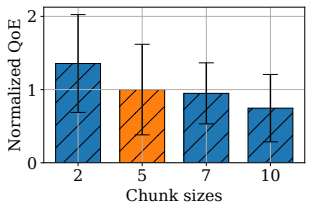


Figure 22: Dashlet's chunk duration's impact on QoE (normalized for 5-second chunk). Bars list averages, with error bars for one st. dev. in each direction.

various number of equal-length chunks. Notice that Tiktok's chunking also leads to a *fix bitrate* for chunks in the same video. *Buffer order:* whether the system follows TikTok or Dashlet's buffer order. *Bitrate selection.* whether the system follows TikTok or Dashlet's bitrate selection. We implement the TikTok's logic for the corresponding design components according to our TikTok analysis in §2.2. For the bitrate selection, we use a lookup table to record the TikTok's bitrate decision under different network throughput and buffer level. Our implementation for TikTok's bitrate choice will then make the decision according to the look-up table.

We first investigate the performance drop when replacing Dashlet's design components with the corresponding TikTok's design component. Fig. 18 shows the QoE difference compared to Dashlet. Dashlet+Prebuffer idle (DID) curve shows that having a prebuffer idle state will have a significant negative impact at low throughput (*e.g.* 0-2 Mbps). But when the impact diminishes as the network throughput is above 4 Mbps. Similarly, TikTok's chunking also has a significant negative impact at the low network throughput. The low network throughput forces TikTok to choose a low video bitrate, but consequently increases the first chunk duration, making TikTok more vulnerable to rebuffering when swipe happens. TikTok's buffering order (DTBO) selection has significant negative impact on QoE until the throughput reaches 14 Mbps. The bitrate selection (DTBS) have the most significant impact on the QoE. Its impact to the QoE dominants as the network throughput reaches 4-6 Mbps. By digging deep in the reason, we find TikTok is very conservative in choosing high bitrate. We show the detail in §C. One natural next question arises is that could we just simply increase the request bitrate to improve the QoE. To answer this question, we consider another ablation study case **TDBS**, which includes TikTok's design for all other components but keeps the high bitrate choices as Dashlet. Fig. 19 shows the comparison between TDBS and TikTok. with the higher bitrate choices, TDBS performs worse than TikTok when the network throughput is less than 12 Mbps. The key reason behind is that TDBS has a higher rebuffer percentage compared with TikTok. The takeaway is that TikTok's low bitrate is a result of adaptation to avoid rebuffering. Simply increase the downloading bitrate could lead to a worse QoE.

5.4 Micro Benchmarks

Impact of Swipe and Network Speeds on QoE. Patterns in network throughput and user swipes largely influence the performance of short video streaming algorithms. To understand the effect of each, we report Dashlet's and TikTok's results for different network throughputs and swipe rates. As shown in Fig. 20, the major factor that affects QoE with Dashlet is the network throughput. Importantly, swipe speed does not have a significant impact on Dashlet's performance, validating its robustness under different swipe patterns. In contrast, both network throughput and swipe speed have a large impact on TikTok's QoE.

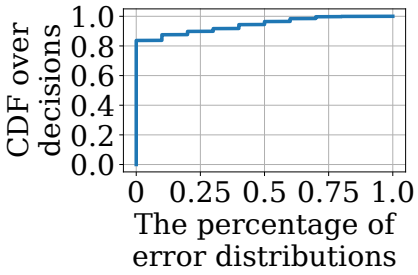


Figure 23: Dashlet's tolerance to swipe distribution errors.

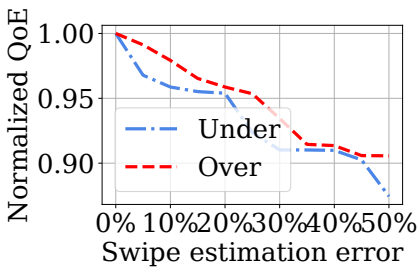


Figure 24: Impact of swipe estimation errors on Dashlet.

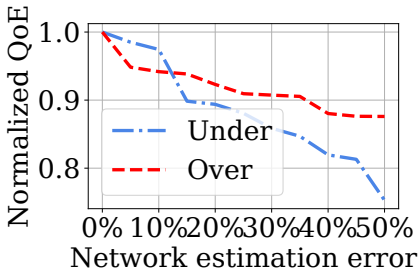


Figure 25: Impact of network estimation errors on Dashlet.

Network Idle and Data Waste. To dig deeper into Dashlet's performance gain over TikTok, we investigate network idling and data wastage for both systems. Fig. 21 shows our results; note that the Oracle algorithm does not incur any data wastage since it has perfect knowledge of user swipe times. As shown, median data wastage and idle time for Dashlet are 29.4% and 45.5%, respectively, which are 30.0% and 35.9% lower than those with TikTok. These improvements enable Dashlet to stream video at higher bitrates than TikTok while keeping rebuffering delays low.

The impact of chunk size on Dashlet's QoE. Unlike TikTok which breaks up videos by bytes, Dashlet (by default) breaks up videos into 5-second chunks. We evaluated the impact that chunk sizes have on Dashlet's performance by considering the following chunk sizes (based on prior work [42]): {2, 5, 7, 10} seconds. Note that we did not modify chunk sizes for TikTok as we could not alter its video servers. As shown in Fig. 22, Dashlet's performance decreases as chunk sizes grow, *e.g.*, average QoE drops by 35.4% as chunk sizes grow from 5 to 10 seconds. The reason is that data wastage grows with larger chunk sizes: a user swipe at 1 second into a chunk will result in more wasted bytes with a larger chunk size.

Decision Stability with Swipe Prediction Errors Dashlet determines buffer sequences by leveraging (coarse information from) users' swipe distributions for each video. Thus, a natural question is how robust are Dashlet's decisions to errors in those distributions, i.e., does it make the right decisions even with different degrees of errors?

Recall that there are three inputs to Dashlet's algorithm at any time: the swipe distribution for each considered video, the estimated network throughput, and the client-side player's current buffer state. The algorithm uses this information and returns a buffer sequence of chunks to download, with the first chunk in the ordered list indicating the action to perform immediately, i.e., the chunk to download now. To answer the above question, we profiled the above inputs throughout our experiments, and then compared the actions selected by Dashlet with those that it would select if the input swipe distribution involved errors. In particular, we considered 10 versions of each video's distribution by (roughly) modeling its original distribution as an exponential one, and then altering the corresponding λ value to change the average swipe time by $1\pm\{0\text{-}50\%\}$ (in 10% increments).

Fig. 23 shows our results. As shown, 83.7% of Dashlet's decisions are unchanged across all of the considered distribution errors. The values remain relatively stable as errors grow – *e.g.*, 96.5% of Dashlet's decisions are unchanged with errors of 50% – but begin to grow after 82%. These results illustrate that Dashlet only relies on coarse information from swipe distributions (*e.g.*, about whether a user is likely to swipe early or late in the video); it is for this reason that decisions are varied only when errors are very high (and even the coarse information that Dashlet uses has changed).

QoE sensitivity with Swipe and Network Errors Building on the previous results, we now analyze how errors in swipe distribution affect the QoE that Dashlet delivers. We ran Dashlet on all videos and the network traces using same faulty distributions from above. Fig. 24 shows the results, breaking them down in terms of scenarios with over estimation of swipe times (longer average viewing time than the correct distribution, i.e., later swipes) and under estimation (shorter average viewing time). As shown, Dashlet is quite tolerant to such errors, delivering 87% and 91% of its full QoE (with no errors) when the traces are over- and under-estimating swipe times by 50%.

We perform a similar analysis to evaluate Dashlet in the presence of network prediction errors. Specifically, we replace the network predictor in RobustMPC [40] with one that reads in the actual instantaneous throughput from the current Mahimahi trace, and multiplies that value by between $1\pm\{0\text{-}50\%\}$. Overall, as per Fig. 25, we find that Dashlet's QoE drops to 88% and 76% of its values without network errors when the network estimate is over- or under-estimating by 50%. These results highlight that Dashlet is more robust to errors in swipe distributions than network forecasts.

6 Related work

Traditional adaptive video streaming Traditional video streaming services deliver video content from the CDN to the user with adaptive bitrate system with the objective of

maximizing the quality of experience for users [20]. Research effort has been made to improve the quality of experience from different perspectives, including streaming algorithm [18, 19, 22, 29, 40], video codec [7, 10], network prediction [30, 36], protocol design [14, 45], and video super resolution [38, 39]. But all these optimization is for the same video streaming model: the video download sequence is the same as the video playing sequence. Dashlet also uses QoE as the optimization goal but tackles a different problem as in the short video streaming the video download sequence is the same as the video playing sequence due to users' swipes.

Streaming new form of video There are also rising interest on 360 degree video [12, 26] and volumetric video streaming [25]. These systems need to handle the uncertainty from the users' head position or location. Dashlet's design also models the uncertainty from the user swipe patterns. But Dashlet targets on a different problem compared with 360 degree or volumetric video streaming. Some existing works [15, 27] also try to apply reinforcement learning algorithms from traditional video streaming [22] to short video streaming. However, these works do not factor in the impact of user swipes on buffering decisions as Dashlet does.

7 Discussion

TikTok version. Our reverse engineering tool can only deciphers the HTTPS messages transmitted by TikTok with version up to v20.9.1. As a result, we cannot conduct our case study (§2) using the up-to-date TikTok v26.3.3, which adopts a different encryption method as V20.9.1. We leave the task of deciphering the HTTPS messages and thus studying the streaming algorithm of the newest version of TikTok as our future work.

Backward swipes, fast-forwarding, and pause. Our current model only allows forward swipes, *i.e.*, swipes to watch next videos. The newest version of TikTok also allows backward swipes where the user swipes to watch the previous video and fast-forwarding, where the user speeds up the playback of the current video: we will study these in future work. In addition, our model does not consider the video pause. The pausing of videos will make it easier for the system since it gives the player more time to download videos. For Dashlet's design, we focus on a harder problem, which assumes no pause in the video.

Diminished gain at higher network speeds. We observe a diminished improvement for Dashlet over TikTok at higher network speeds. At higher network speeds, mistakes made by TikTok are masked by higher network throughput. As network speed increases, TikTok can pick up the highest bitrate but still have enough time to react to users' wipes. In our evaluation, we use the bitrates that TikTok's CDN offers, which are capped at 720P video quality. We expect the gap between Dashlet and TikTok will widen if higher bitrate videos are used to evaluate both systems, which we expect will happen in the future.

Generalization of Dashlet design. The Dashlet design does not rely on the design of TikTok but only relies on a sequence of videos that are played in chunks. Therefore, it should be able to generalize to other platforms like YouTube Shorts and Instagram Reels.

Energy implication to smartphones. Dashlet could potentially reduce the energy consumption for short video applications. The energy cost includes both the cost to run the algorithm and the cost to download the video content. Dashlet uses a simple non-machine learning algorithm, which causes minimal extra energy overhead. For the cost to download the video content, Dashlet has less energy overhead since its waster download is much less than TikTok.

Evaluation generalizability. We have conducted a small scale human study to compare the performance of Dashlet and TikTok, where ten participants log into their own account to watch TikTok video on a emulated mobile network, repeating the experiments using Dashlet. The personality of the recruited user may lead to biased results, for example, a patient user may tend to not swipe or swipe at the end of the video, leaving larger time window for TikTok to download the second chunk. A larger scale human study that involves more diverse users is needed to eliminate this potential bias.

We conduct our evaluation under emulated mobile networks, but prior work [36, 37] has pointed out that the emulation based evaluation of network applications and congestion control schemes may not always be indicative of real-world performance. For example, although we compensate the average round trip delay to the CDN server in the emulated environment, the variation in the round trip delay might potentially impact the results. While we note that Dashlet does not input network measurements into an ML model, we acknowledge that large-scale evaluation in the wild may be required to verify the full generalizability of our results.

8 Conclusion

In this paper, we design and implement Dashlet with the insight provided by measurement for a commercial short video app and a user study on general user swipe pattern. Dashlet's algorithm strategically determines the buffer order with the input from a coarse-grained swipe distribution. Evaluation result shows Dashlet significantly improves video quality and reduces rebuffering compared with the baseline system.

9 Acknowledgement

We thank the anonymous reviewers and our shepherd, Sanjay Rao for their insightful comments. This material is supported in part by NSF CNS grants 2152313, 2153449, 2147909, 2140552, and 2151630, Sloan Research Fellowship, as well as a grant from the Princeton School of Engineering and Applied Science.

References

[1] 24 Important TikTok Stats Marketers Need to Know in 2022. https://blog.hootsuite.com/tiktok-stats/.

[2] AKHTAR, Z., NAM, Y. S., GOVINDAN, R., RAO, S., CHEN, J., KATZ-BASSETT, E., RIBEIRO, B., ZHAN, J., AND ZHANG, H. Oboe: Auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2018), SIGCOMM '18, Association for Computing Machinery, p. 44–58.

[3] CHEN, F., LI, P., ZENG, D., AND GUO, S. Edge-assisted short video sharing with guaranteed quality-of-experience. *IEEE Transactions on Cloud Computing* (2021), 1–1.

[4] CHEN, Z., HE, Q., MAO, Z., CHUNG, H.-M., AND MAHARJAN, S. A study on the characteristics of douyin short videos and implications for edge caching. In *Proceedings of the ACM Turing Celebration Conference-China* (2019), pp. 1–6.

[5] CORTESI, A., HILS, M., KRIECHBAUMER, T., AND CONTRIBUTORS. mitmproxy: A free and open source interactive HTTPS proxy, 2010–. [Version 8.0].

[6] DASARI, M., BHATTACHARYA, A., VARGAS, S., SAHU, P., BALASUBRAMANIAN, A., AND DAS, S. R. Streaming 360-degree videos using super-resolution. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications* (2020), IEEE Press, p. 1977–1986.

[7] DASARI, M., KAHATAPITIYA, K., DAS, S. R., BALASUBRAMANIAN, A., AND SAMARAS, D. Swift: Adaptive video streaming with layered neural codecs. In *USENIX NSDI* (2022).

[8] DASH Industry Forum. https://reference.dashif.org/dash.js/latest/samples/index.html.

[9] Federal Communications Commission. 2016. Raw Data - Measuring Broadband America. (2016). https://www.fcc.gov/reports-research/reports/.

[10] FOULADI, S., EMMONS, J., ORBAY, E., WU, C., WAHBY, R. S., AND WINSTEIN, K. Salsify:{Low-Latency} network video through tighter integration between a video codec and a transport protocol. In *USENIX NSDI* (2018).

[11] GAO, X., LE CALLET, P., LI, J., LI, Z., LU, W., AND YANG, J. *QoEVMA'20: 1st Workshop on Quality of Experience (QoE) in Visual Multimedia Applications*. Association for Computing Machinery, New York, NY, USA, 2020, p. 4771–4772.

[12] GUAN, Y., ZHENG, C., ZHANG, X., GUO, Z., AND JIANG, J. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *ACM SIGCOMM* (2019).

[13] GUO, J., AND ZHANG, G. *A Video-Quality Driven Strategy in Short Video Streaming*. Association for Computing Machinery, New York, NY, USA, 2021, p. 221–228.

[14] HAN, B., QIAN, F., JI, L., AND GOPALAKRISHNAN, V. Mp-dash: Adaptive video streaming over preference-aware multipath. In *CoNEXT* (2016).

[15] HE, J., HU, M., ZHOU, Y., AND WU, D. Liveclip: towards intelligent mobile short-form video streaming with deep reinforcement learning. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video* (2020).

[16] HUANG, T.-Y., JOHARI, R., MCKEOWN, N., TRUNNELL, M., AND WATSON, M. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, Association for Computing Machinery, p. 187–198.

[17] ITSEEZ. *The OpenCV Reference Manual*, 4.5.1 ed., 2022.

[18] JIANG, J., SEKAR, V., AND ZHANG, H. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *ACM CoNEXT* (2012).

[19] KIM, J., JUNG, Y., YEO, H., YE, J., AND HAN, D. Neural-enhanced live streaming: Improving live video ingest via online learning. In *ACM SIGCOMM* (2020).

[20] KRISHNAN, S. S., AND SITARAMAN, R. K. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. *IEEE/ACM Transactions on Networking* (2013).

[21] LYMBEROPOULOS, D., RIVA, O., STRAUSS, K., MITTAL, A., AND NTOULAS, A. Pocketweb: Instant web browsing for mobile devices. In *ASPLOS 2012 (Architectural Support for Programming Languages and Operating Systems)* (March 2012), ACM.

[22] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM* (2017).

[23] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate {Record-and-Replay} for {HTTP}. In *USENIX ATC* (2015).

[24] PyAutoGUI - PyPI. https://pypi.org/project/PyAutoGUI/.

[25] QIAN, F., HAN, B., PAIR, J., AND GOPALAKRISHNAN, V. Toward practical volumetric video streaming on commodity smartphones. In *ACM HotMobile* (2019).

[26] QIAN, F., HAN, B., XIAO, Q., AND GOPALAKRISHNAN, V. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *ACM MobiCom* (2018).

[27] RAN, D., ZHANG, Y., ZHANG, W., AND BIAN, K. Ssr: Joint optimization of recommendation and adaptive bitrate streaming for short-form video feed. In *IEEE MSN* (2020).

[28] Genymobile/scrcpy: Display and control your Android device. https://github.com/Genymobile/scrcpy.

[29] SPITERI, K., URGAONKAR, R., AND SITARAMAN, R. K. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking 28*, 4 (2020), 1698–1711.

[30] SUN, Y., YIN, X., JIANG, J., SEKAR, V., LIN, F., WANG, N., LIU, T., AND SINOPOLI, B. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *ACM SIGCOMM* (2016).

[31] TikTok: Thanks a billion! https://newsroom.tiktok.com/en-us/1-billion-people-on-tiktok.

[32] TikTok Revenue and Usage Statistics (2022). https://www.businessofapps.com/data/tik-tok-statistics/.

[33] https://influencermarketinghub.com/tiktok-stats/. https://influencermarketinghub.com/tiktok-stats/.

[34] TikTok User Statistics (2022). https://backlinko.com/tiktok-users.

[35] WANG, S. W., YANG, S., LI, H., ZHANG, X., ZHOU, C., XU, C., QIAN, F., WANG, N., AND XU, Z. Salientvr: Saliency-driven mobile 360-degree video streaming with gaze information. In *Proceedings of the 28th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2022), MobiCom '22, Association for Computing Machinery.

[36] YAN, F. Y., AYERS, H., ZHU, C., FOULADI, S., HONG, J., ZHANG, K., LEVIS, P., AND WINSTEIN, K. *Learning in Situ: A Randomized Experiment in Video Streaming*. USENIX Association, USA, 2020, p. 495–512.

[37] YAN, F. Y., MA, J., HILL, G. D., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon: the training ground for internet congestion-control research. In *USENIX ATC* (2018).

[38] YEO, H., CHONG, C. J., JUNG, Y., YE, J., AND HAN, D. Nemo: enabling neural-enhanced video streaming on commodity mobile devices. In *ACM MobiCom* (2020).

[39] YEO, H., JUNG, Y., KIM, J., SHIN, J., AND HAN, D. Neural adaptive content-aware internet video delivery. In *USENIX OSDI* (2018).

[40] YIN, X., JINDAL, A., SEKAR, V., AND SINOPOLI, B. A control-theoretic approach for dynamic adaptive video streaming over http. In *ACM SIGCOMM* (2015).

[41] YouTube Shorts Video-Making App Now Receiving 3.5 Billion Daily Views. https://www.latestly.com/technology/youtube-shorts-video-making-app-now-receiving-3-5-bil html.

[42] ZHANG, T., REN, F., CHENG, W., LUO, X., SHU, R., AND LIU, X. Modeling and analyzing the influence of chunk size variation on bitrate adaptation in dash. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications* (2017), IEEE, pp. 1–9.

[43] ZHANG, X., OU, Y., SEN, S., AND JIANG, J. SENSEI: Aligning video streaming quality with dynamic user sensitivity. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association, pp. 303–320.

[44] ZHANG, Y., LIU, Y., GUO, L., AND LEE, J. Y. Measurement of a large-scale short-video service over mobile and wireless networks. *IEEE Transactions on Mobile Computing* (2022).

[45] ZHENG, Z., MA, Y., LIU, Y., YANG, F., LI, Z., ZHANG, Y., ZHANG, J., SHI, W., CHEN, W., LI, D., ET AL. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *ACM SIGCOMM* (2021).

A Dashlet Pseudocode

Algorithm 1: Dashlet's ABR algorithm

Input : 1) Buffer status $r_1 \dots r_n$
2) Video bitrate $B = \{b_{11} - b_{1k} \dots b_{n1} - b_{nk}\}$
3)The probability distribution of play start time for each chunk $\hat{G} = \hat{g}_{11}(t) \dots \hat{g}_{nc_n}(t)$
4) network throughput estimation T
5) The look-ahead time length F
6) Chunk length L

Output :The location and bitrate to buffer next

foreach $i, j \in \{1 \dots n\}$ **do**
  if $\int_0^F (F-t)\hat{g}_{c_{ij}}(t)dt > 1/\mu$ ***and*** $j > r_i$ **then**
    $candidateList.append(c_{ij})$;
  //Add the chunk to the candidate list if there is significant penalty for not downloading it
$targetBitrate = F \times T$ / $len(candidateList)$ / L;
do
  c_{min} = `minRebufferCost` ($targetBitrate$, $bufferOrder$, $candidateList$);
  $bufferOrder.append(c_{min})$;
  $candidateList.remove(c_{min})$;
while $len(candidateList) > 0$;
//use greedy algorithm to put chunks from *candidateList* into *bufferOrder*
$bitrateList$ = `getMaxBitrate` ($bufferOrder$, B, T);
//Enumerate all the bitrate combination for chunks in *bufferOrder* to maximize the QoE
Return $bufferOrder[0]$, $bitrateList[0]$

B Dashlet Implementation Further Detail

Dashlet makes no change to the CDN/server side so our system can be easily deployed client side. Dashlet includes one control module and multiple buffer modules. Each buffer module manages the video playback of one short video, including downloading chunks, tracking playback progress, and reporting buffer status. We reuse the DASH.js playback management for the buffer modules. The control module manages scheduling across short videos, collecting estimated throughput and buffer length from each buffer module. With the collected data, control module runs Dashlet's algorithm to schedule the video buffering. Based on the algorithm's output, it assigns the quota to the buffer module that is assigned to download the next video chunk. The quota includes the target video bitrate and the target download finish time. Once the buffer module receives the quota, it sends an HTTP request with target bitrate to the CDN to download s the corresponding video chunk. A call back function is set to report the status to control module in case the download time exceeds the target download finish time. The control module schedules the video buffering when the call back function for target download time is triggered, the chunk download finishes, or the user swipes. Similar to Pensieve [22], we also use an ABR server to run Dashlet's algorithm on the same machine as the client. The control module communicates with the ABR server using XMLHttpRequests locally.

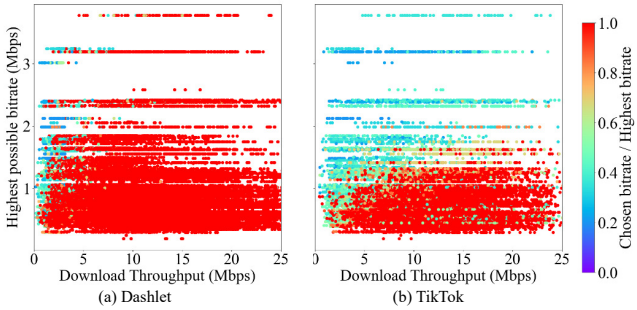


Figure 26: Bitrate choice made by Dashlet and TikTok. The x-axis is the network throughput and the y-axis is the highest available bitrate to choose. The color is the ratio between the chosen bitrate and the highest available bitrate. The red color means the highest available bitrate is chosen.

1. How will you rate the quality of the video you watched on **TikTok**
   a. Excellent, all the videos are high quality (5)
   b. Good, the video quality is generally good (4)
   c. OK, there are some blur videos but do not affect my watching experience (3)
   d. Need improve, there are a lot of blur videos and affect my watching experience (2)
   e. Bad, the videos are not clear at all (1)

2. How will you rate the quality of the video you watched on **Dashlet**
   a. Excellent, all the videos are high quality (5)
   b. Good, the video quality is generally good (4)
   c. OK, there are some blur videos but do not affect my watching experience (3)
   d. Need improve, there are a lot of blur videos and affect my watching experience (2)
   e. Bad, the videos are not clear at all (1)

3. How will you rate the smoothness of the video you watched on **TikTok**
   a. Excellent, I did not experience any rebuffer (5)
   b. Good, the video plays smoothly except only a few short rebuffer (4)
   c. OK, the video stalls sometime, but it does not affect my watching experience (3)
   d. Need improve, the video stalls a lot, it affects my watching experience (2)
   e. Bad, the video has lots of stall and can hardly be played (1)

4. How will you rate the smoothness of the video you watched on **Dashlet**
   a. Excellent, I did not experience any rebuffer (5)
   b. Good, the video plays smoothly except only a few short rebuffer (4)
   c. OK, the video stalls sometime, but it does not affect my watching experience (3)
   d. Need improve, the video stalls a lot, it affects my watching experience (2)
   e. Bad, the video has lots of stall and can hardly be played (1)

Figure 27: Questionnaire for user survey.

C TikTok is conservative in video bitrate selection.

We show every bitrate that TikTok and Dashlet has selected in the section with Fig. 26. We can conclude from the figure that TikTok limits its bitrate even if the network throughput is high enough. This then leads to a significant negative impact on the QoE.

D Questionnaire sample.

We show the sample of the questionnaire we used in the user survey in Figure 27.