Improving Quality of Experience for Mobile Broadcasters in Personalized Live Video Streaming

Anonymous

Abstract

I. Introduction

Recent years have seen the coming of age of personalized live streaming. With more personal devices equipped with high-definition cameras, we observe a rapid proliferation of apps that allow users to stream videos from their smartphones or tablets to anyone who tunes in. Such personalized live streaming has found its world-wide popularity as a way of engaging with more followers (e.g., Twitter Meerkat [6], iQIYI [5]), sharing richer experience (e.g., Facebook Live [1], Periscope [3]), and broadcasting online gaming and sports events (e.g., Twitch [4], Douyu [2]).

While recent work on personalized live streaming has insofar focused on its traffic pattern (e.g., [15], [12]) and video distribution architecture (e.g., [10], [13]), not enough efforts have been made to characterize the quality issues of broadcaster-uploaded videos in the wild among popular platforms of personalized live streaming. We argue that understanding and improving the broadcaster-side video quality is crucial to the Quality of Experience (QoE) of personalized live streaming for two reasons:

1. Broadcaster-side quality issues have a direct impact on *all* viewers: any delay or failure caused by the broadcaster could inflate the streaming delay of all viewers, and the upstream video quality sets a "cap" on the QoE of all viewers (even if they have high-speed downlink connections). As a result, for instance, we observe the broadcasters typically only upload videos in the highest constant bitrate.
2. Unlike traditional live streaming of popular events (e.g., ESPN) where broadcasters have well-provisioned connections and streaming delay is typically at least 10 seconds, personalized live streaming poses new challenges on broadcaster-side QoE, because (a) the broadcasters are mobile users with highly variable network performance due to wireless packet losses and user mobility, and (b) the end-to-end streaming delay must be several seconds to create real-time interactivity when the broadcaster interacts with viewers who pose questions or "likes".

Despite its significance, today's QoE of broadcaster-uploaded video, according to our experiments, is not desirable and we observe two *prevalent* quality issues across many popular platforms in the wild. In particular, we observe an "amplifying effect" of network condition on video QoE: *a transient throughput degradation of less than a second on the broadcaster side can lead to several seconds of video stall observed by the viewers*. Besides, *long-term network fading is not rare in real world*. Such problem is particularly prevalent, because the broadcasters (e.g., smartphones, tablets) are often wireless-connected and, both long-lived degradation and transient throughput drops are not uncommon [**?**] in today's wireless networks, due to cellular hand-off, WiFi-cellular switch, device moving-around and so forth.

The root cause of this broadcaster-side "amplifying" quality problem lies in the fact that RTMP, the de-facto broadcaster-side streaming protocol, drops video frames too aggressively when video buffer overflow occurs, resulting in unnecessary drops of important video frames and persistent video stalls on the viewer side. Moreover, intuitive strawman solutions, such as increasing buffer length, alternative frame-dropping policies) fail to meet at least one of the two QoE requirements of personalized streaming: they either increase end-to-end delay (i.e., low timeliness), or drop more frames than needed (i.e., low video resolution). For instance, simply increasing buffer size on the broadcaster side hides transient throughput drops but may cause end-to-end delay to grow unboundedly.

For the long-term network drops, state-of-art related works focus on the player strategy in DASH. Little try is put into the research of broadcaster' side. Using the complete MPC is one possible solution, but different from DASH, in live video streaming, buffer is little to have improve space. Besides, RobustMPC takes long time to calculate and FastMPC may introduce some quantization error.

We argue that such broadcaster-side quality issues are reduced significantly by a systematic design of RTMP configuration (i.e., key frame interval, buffer size) and logic (i.e., frame-dropping policy) that take both video resolution and timeliness as objectives. Our preliminary evaluation shows that a better RTMP design could significantly improve video quality compared to three popular RTMP-based commercial platforms as well as an open-source RTMP platform. Furthermore, we found that a greedy bitrate adaptation algorithm can perform well enough in live streaming scenario, which is an interesting found. The proposed greedy algorithm, which is called GA, can reduce the frame dropping into an acceptable level(cut down 80% of the frame dropping) and keep the original bitrate at the same time.

This paper, we make two contributions:

1) We are the first to shed light on the broadcaster-side video quality issues across three today's personalized streaming platforms and identify its root cause.
2) Propose three principles to guide the broadcaster de-

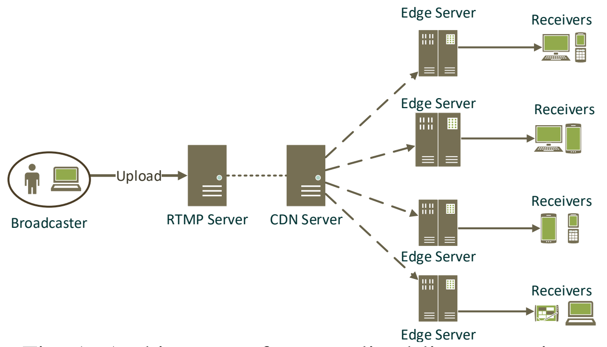


Fig. 1: Architecture of personalized live streaming.

sign, and for each item, give detailed formulation.

3) Show both qualitatively and quantitatively that there is a significant room for improving the broadcaster-side video quality by a better design of frame drop scheme and bitrate adaptation algorithm.

II. Background

We start with the background of personalized live streaming, including its similarities and key differences to traditional live streaming, and what is their subsequent implication on the system implementation, especially on broadcaster-side streaming protocol.

A. Overview of architecture

Figure 1 shows the common architecture of most popular personalized live streaming platforms. When live streaming starts, the broadcaster uploads the live video to an edge server using RTMP protocols, where the video is further forwarded to an entry server of a CDN After that, the CDN uses its overlay networks to distribute the video to many edge servers. Finally, each viewer streams the video from a nearby edge server using HTTP-based streaming protocols (i.e., DASH [**?**]).

B. Personalized live streaming vs. other live streaming

First of all, both personalized live streaming and traditional live streaming (e.g., ESPN's sports broadcast and CNN's scheduled programs) share the need to distribute video content to a large audience at a low cost. Therefore, both types of live streaming rely on existing CDN infrastructure to distribute video content to viewers through HTTP-based streaming protocols.

Despite the similarities, personalized live streaming has two key differences:

Mobile users as broadcasters: Traditional live streaming (e.g., ESPN) uses a dedicated over-provisioned connection (usually direct cable or an exclusive satellite channel) to stream high-resolution raw video content from a camera to a special content management server which transcodes the video from the original forms to video chunks that can be efficiently distributed to edge servers. In contrast, the content source in personalized video streaming is often mobile users who upload the live video through a wireless connection shared with many other users.

Broadcaster-viewer interactivity: Another key difference is that ensuring low end-to-end streaming delay is critical in personalized live streaming for broadcasters to interact with viewers, where viewers in traditional sports live events only passively watch the video. For instance, the broadcaster may want to thank the audience instantaneously if he/she is given gifts from a viewer; in game streaming, the broadcaster may make frustrating mistakes without instantaneous feedback from the viewer. Therefore, the streaming delay ideally should be no more than several seconds, which is much less than traditional sports live streaming.

C. Broadcaster streaming protocols

These differences lead to three requirements (formally defined in the next section) on how the broadcaster streaming protocol:

1. *High bitrate:* The broadcaster must encode the video in high bitrate and send the high-bitrate video to the edge server, so that downstream viewers can watch the video in the best possible resolution.
2. *Agility:* The streaming protocol must be sufficiently adaptive to quickly react the performance fluctuations in wireless networks.
3. *Timeliness:* The streaming protocol must ensure the streaming delay between the broadcaster and viewers is minimized or at least bounded.

For practical reasons, RTMP has become the de-facto broadcaster streaming protocol in most of today's platforms, including Facebook Live, Twitch, Periscope, iQIYI, Douyu, and so forth. RTMP is flexible enough to potentially meet the three aforementioned requirements. For instance, it offers several tunable parameters for the broadcaster to adjust the video quality, including frames per second (FPS), buffer size, and frame dropping policy. In theory, the frame-dropping policy could strike a dynamic balance between quality and timeliness in the presence of throughput fluctuation (e.g., [7], [8], [11]). Nonetheless, as we will show in the next section, both commercial implementations of RTMP and the up-to-date open source RTMP implementation suffer from similar quality degradation.

Alternative HTTP-based broadcaster streaming protocols have also been studied, including using DASH [**?**], HTTP POST [9], and adaptively switching between them [14]. While switching from RTMP to HTTP-based protocols might achieve better video quality, it requires costly changes on client-side software and cannot react to wireless fluctuation in a timely manner due to chunking overheads (each chunk is at least of several seconds).

III. Measurement and Analysis

A. Motivating Examples

Experiment setup. We set up a live video streaming framework as in Figure 2. Two servers are connected to a

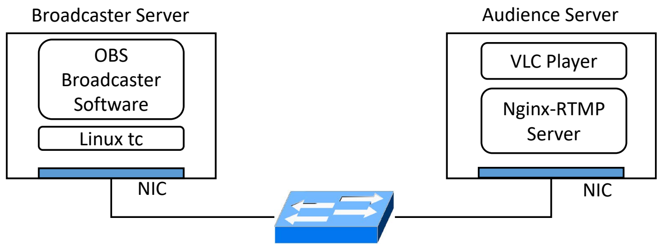


Fig. 2: Experiment setup

switch in the middle. The servers have 2 CPU cores and 6GB memory, and are equipped with 100Mbps NICs. OBS studio (sender) is used to stream videos to the audience side over RTMP protocol, the broadcaster server (receiver) is built on nginx-rtmp module. On the audience server (receiver), we use the VLC player to play the video. In the experiment, we use tc [?] to control the real-time bandwidth on the sender side.

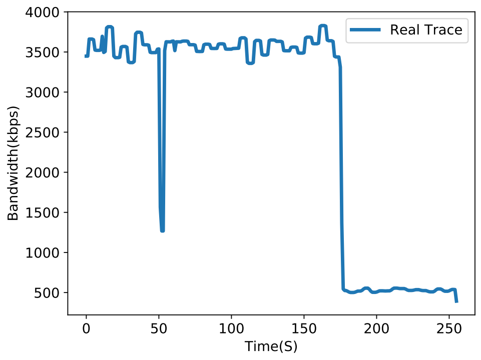
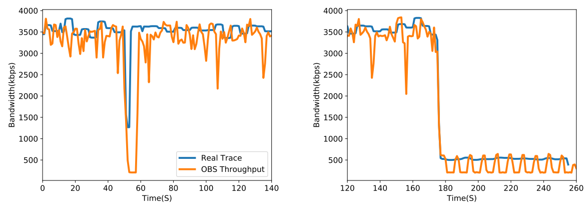


Fig. 3: Bandwidth Control

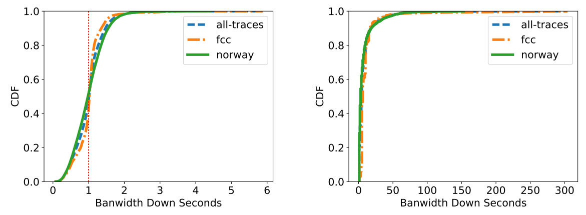


(a) Actual throughput of $0-150$s (b) Actual throughput of $120-260$s

Fig. 4: Case study: video streaming throughput in oscillating wireless network

Case study. In the first motivating experiment, we control the network bandwidth according to an actual trace from a wireless network. The trace is from FCC [?] and describes the network conditions when a user join the www.amazon.com on a mobile device. We aggregate packets in the trace into 5-second bins and calculate the data amount in each bin. We then control the network bandwidth on the sender side according to the per-second profile. The network bandwidth is shown in Figure 3. Meanwhile, we stream video at a bitrate of 3000kbps via OBS and capture actual video packet trace using tcpdump. We aggregate packets in the video packet trace into bins of 1 second. And the result is shown in Figure 4.

In the figure 4b, the actual throughput roughly equals to the minimum of the controlled bandwidth and the choosen bitrate. However, we observe that at 50s, the network bandwidth falls below the bitrate and the situation lasts until 53, while the actual throughput degrades below the bitrate from 50s to 58s (except a small spike). This is an abnormal behavior, as *a 2-second network jitter cascadingly causes 8-second throughput falling in the streaming application.* Besides, a constant bitrate cannot efficiently handle long-term the bandwidth variance, which can be seen in Figure 4a. Bandwidth is enough during $0-180s$, but after $180s$, the available bandwidth drops dramatically and lasts for $80s$. In this challenging network environment, the default The OBS won't low down its bitrate and causes tremendous frame dropping.



(a) Normalized Bandwidth Distribution (b) Bandwidth Down Time Distribution

Fig. 5: Trace analysis: Bandwidth down in wireless network

Network Conditions To know the realistic network conditions, first we want to know the distribution of the traces. Using the average bandwidth as the unit, we normalize other value into the relative delta(Figure. 5a). We combine several public datasets: FCC traces and HSDPA traces. The total trace lasts for 30 hours. Almost 50% of traces are under the average throughput, which is shocking. About 20% of the traces are at most half of the average. The first picture indicates that in real-time network, bandwidth fluctuation frequently occurs. To further explain how frequently bandwidth fluctuation happens, we draw a picture of bandwidth down distribution, in Figure 5b. The value is calculated by counting the continuous time lower than the average bandwidth. About 20% of the bandwidth fluctuation lasts for more than 10 seconds, some even lasts for hundreds of seconds. Always using constant bitrate may introduce massive frame dropping.

Experiments on several commercial platforms. We further repeat the experiment in different commercial platforms and settings, including OBS pushing video to Douyu server, Douyu broadcaster to Douyu server, and OBS to twitch server. Figure 6, 7, and 8 show the throughput of the three

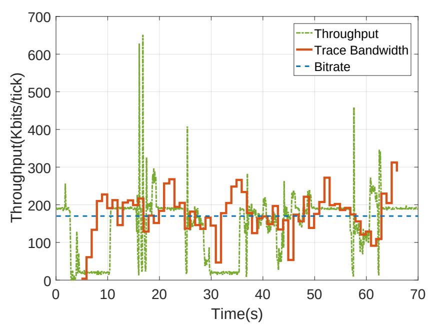


Fig. 6: Throughput, OBS to Douyu server

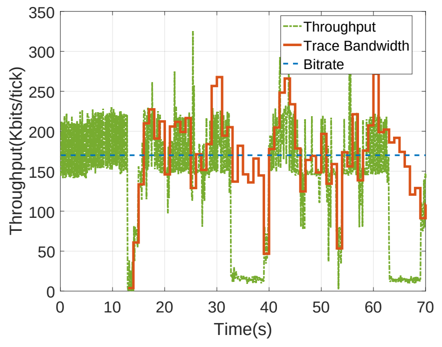


Fig. 7: Throughput, Douyu broadcaster to Douyu server

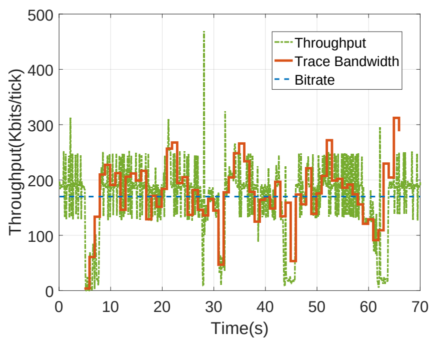


Fig. 8: Throuhput, OBS to Twitch server

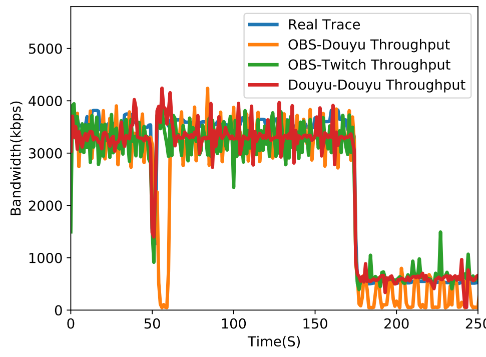


Fig. 9: Bandwidth Control

TABLE I: Frames dropped in different scenarios

Scenario	Seconds of Black Screen(S)	Percentage(%)
$Obs-douyu(a)$	18.1	30.2%
$Obs-twitch(a)$	9.9	16.3%
$Douyu-douyu(a)$	16.6	27.2%
$Obs-douyu(b)$	93.1	37.2%
$Obs-twich(b)$	79.3	31.7%
$Douyu-douyu(b)$	66.67	26.7%

experiments respectively, and Figure show the corresponding frame drops during the experiment.

Comparing the results across different platforms, we observe that the "cascading effect" is prevalent, appearing on all platforms (e.g., the 30s in OBS to Douyu, the 32s in Douyu to Douyu, and 43s in OBS to Twitch). We also find out that the cascading effect is not related to the instantaneously available bandwidth. For example, in Figure 6, a dramatic bandwidth drop at 30s causes the cascading frame drop; while in Figure 7, a slight bandwidth drop at 32s causes the frame drop. Another observation is that the length of the cascading drop is different on different platforms: ¿5s in Douyu and 2-3s in Twitch. Finally, from Figure, we observe that the broadcaster software usually can tolerate short-period throughput drop without dropping frames, but cannot tolerate long-period ones.

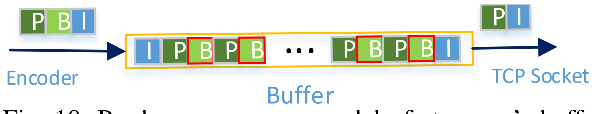


Fig. 10: Producer-consumer model of streamer's buffer

Frames	I	B	B	P	B	B	P	B	B	I	...
Display order	1	2	3	4	5	6	7	8	9	10	...
Coding order	1	3	4	2	6	7	5	9	10	8	...

Fig. 11: H.264 frame display/coding order

B. Analyzing the Root Cause

The cause of "frame drop" is the buffer management in the streaming software. There exists a queue to temporarily store video frames; a video frame generating thread captures images from the camera, encodes raw images into H.264 frames, and enqueues the H.264 frames; while a frame sending thread dequeues frames and send them to the network via TCP socket operations (e.g., `write()`). If the network is in bad conditions, the frame sending thread would be blocked, and then the queue accumulates until a threshold, causing the frame generating thread unable to enqueue frames and thus dropping them.

The cause of the "cascading" drop is the dependency between frames. In H.264, a piece of video is organized into groups of pictures (GOP). During the encoding, the first frame in each group is kept unchanged (I frame); a few P frames are generated by computing their delta with the preceding I or P frame; a B frame is computed based on its neighboring I and/or P frames. Figure 11 shows an example of a series of I, B, P frames. The frames are indexed by display order, but the encoding/decoding is in a different order according to the dependency. Due to the dependency, when a P frame in the middle of a GOP is dropped, all following P, B frames within the same group would not be able to decode. Thus, if a small interruption from the network causes frame drop in the beginning or middle of a group, it cascading causes the remaining frames in the same group not decodable (or simply dropped).

We studied OBS broadcaster software and list its frame

Algorithm 1 OBS Frame Enqueue Management

```
1:  Input: frame
2:  T1 := 0.9s, T2 := 0.7s
3:  if frame is I frame then
4:      dropPFrame := False, dropBFrame := False
5:      ENQUEUE(queue, frame), return
6:  else
7:      timespan := TIMESPAN(queue)
8:  if frame is P frame then
9:      if dropPFrame or timespan > T1 then
10:         DROP(frame), DROP(queue, 'P')
11:         dropPFrame := True
12:     else
13:         ENQUEUE(queue, frame)
14: else if frame is B frame then
15:     if dropBFrame or timespan > T2 then
16:         DROP(frame), DROP(queue, 'B')
17:         dropBFrame := True
18:     else
19:         ENQUEUE(queue, frame)
```

management algorithm (Algorithm 1). At first, the drop priority are set to false. When a new frame arrives at the queue, if it is I frame, it is enqueued (never dropped); otherwise, the timespan of the frames in the queue is computed (i.e., the difference of the display timestamps between the latest and the earliest frame). If the incoming frame is a P frame, and if the timespan is smaller than 0.9 second, the P frame is enqueued; but if the drop priority corresponding P frame is true, the P frame is dropped; and if timespan is larger than 0.9 second, all P and B frames (including the ones in the queue and incoming ones) within the GOP are dropped, all the drop priority are set to true. Similarly, if the incoming frame is a B frame, the threshold is 0.7 second, and the processing logic is the same with that of P frames.

IV. Design Space

To meet the delay constraints, there are two kinds of solution. One is to limit the size of the sender buffer, strictly restrain each frame to meet the requirements; the other maybe controlled by scheduling to maintain the average of delay at the target value. In our case, the first one is our choice. We limit the buffer size to $0.9s$.

With buffer size limited, the previous case study gives us three intuitions to improve the video streaming quality.

A. Design Space Insight

Eliminate the dependency between frames. By this means, the solution space would be larger and more optimal solutions are expected to be found.

There are two ways to implement this constraint relaxation. A naive approach is to reduce the keyframe interval. For example, if a 2-second interruption starts at the beginning of an 8-second GOP, the whole group are dropped; but if the 8-second GOP is refined to be four 2-second GOP, only one 2-second GOP would be dropped. Thus, the cascading effect would be eliminated. However, this approach may be a tradeoff between the minimal frame drop and the video quality, because reducing keyframe interval means less compression in video streaming, to keep a pre-configured bitrate, per-image quality would be degraded (i.e., "big pixels"). The method needs a good tradeoff between video quality and frame dropping.

Another approach is to make the GOP selection adapt to the network condition. In details, when the network recovers from an interruption, the first frame transmitted is encoded as I frame, and a new GOP restarts from this first frame. In this way, the new GOP has no dependency with previous (possible dropped) frames, and all its frames are decodable. This approach may need to modify the encoding workflow, which is hard and out of control.

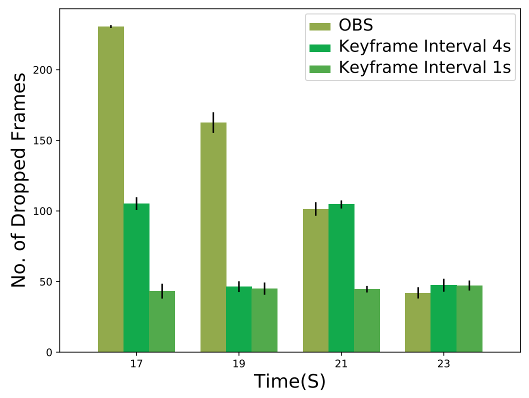
Improve the frame drop strategy. Compared with the naive strategy in OBS (dropping all P/B in buffer when exceeding a threshold), intuitively, dropping frames within the old GoP, rather than all, may have better performance. It is worth thinking how to design an online frame dropping strategy that approaches the optimal solution. The challenge is the complexity of the frame management mechanism. A brute force solution is impossible because its time complexity.

Adaptive bitrate. Network condition is always changing along time. Conclusions from figure 5 validate the fact. Previous broadcasters can only use constant bitrate, CBR or ABR, which means the actual bitrate varies among the target value, at most down or up 20% of the target bitrate. These two methods cannot follow the changing bandwidth, which would bring tremendous frame dropping when bandwidth falls down. Especially in the case where the bandwidth drop lasts for a long while. One possible solution is like DASH in VOD scenario, we use adaptive bitrate in broadcaster's side. Introducing bitrate adaptation maybe dramatically cut down the frame dropping.

B. Preliminary Evaluation

Implementation and experiment setup. In this section, we evaluate two methods: reducing keyframe interval, and changing the bitrate. We design control experiments, where we control the bitrate to a certain level, and introduces a 2-second interruption. We record the number of frame drop as metrics to evaluate these methods.

Varying key frame interval. OBS has a default I frame interval of 8 seconds, and we adjust it to be 4s and 2s in experiments. The frame drop are shown in Figure 12a. We can observe that in each individual experiment, when the interruption starts earlier in a GOP, more frames are dropped, because an early frame has more following frames depending on it. For example, in OBS, when interruption starts at 17s, 19s, 21s, and 23s, the number of frame drop is xx, xx, xx, and xx. Also, the number of frame drop appears to have the same period with the keyframe interval (e.g., when keyframe



(a) Frame drop with varying I frame interval

interval is 4s, the number of frame drop is xx, xx, xx, and xx when interruption starts at 17s, 19s, 21s, and 23s, showing a period of 4s.).

Comparing bars within the group of 17s, we find that smaller keyframe interval significantly reduces the number of frame drop (i.e., from xx, to xx, and xx when the interval is from 8s to 4s and 2s). However, this reduction is not significant for the group of 23s, because 23s is near the end of a GOP in all cases (8s, 4s, and 2s interval), there are only 1-second frames depending on the frame at 23s.

This experiment shows that if we can eliminate the dependency between frames, an occasional network jitter would only affect frames within a limited duration near the jitter, not cascadingly affecting frames in following several seconds. In practical use, reducing keyframe interval is an intractable issue because that adjust would cause video quality degradation. A tradeoff between video quality degradation and frame dropping needs to seriously solved.

TABLE II: No. of Dropped Frames

Bitrate(kbps)	1000	1500	2000	2500
Average Dropped Frames	148.2	148.2	149.0	150.6

Varying bitrate. To make the conclusion more visible, we fix key frame interval to be 8s and introduce network interruption between 19s and 21s. In different experiments, we provide sufficient network bandwidth and vary the bitrate to be 1000kbps, 1500kbps, 2000kbps, and 2500kbps. The frame drop is shown in Table II. The different bitrates do not make much difference, the number of drop in all cases is about 149.

Summary. We summarize and get conclusions. First, reducing keyframe interval leads to less frame drop. Second, bitrate does not influence frame drop for the short-term case, but the quality of each picture. Preliminary Evaluation points out that a small GoP is one useful try.

V. Our Solution and Evaluation

For the three issues, we will propose available solution in this section.

TABLE III: Terminology in Integer Program

Symbol	Type	Meaning
i	index	frame index
j	index	time index
x_{ij}	variable	whether frame i is in queue at time j
y_{ij}	variable	whether frame i is sent at time j
z_{ij}	variable	whether frame i is dropped at time j
T	const	decision time
T_1	const	max time when a frame keeps “fresh”
C_j	const	network bandwidth at time j
N	const	I frame interval
S	const	each frame size
M_j	const	frame index that can be send at time j
R_i	const	bitrate of the i frame

maximize $\Sigma_i y_{iT}$, subject to	
$x_{ij}+y_{ij}+z_{ij}=0, \forall j<i$	(1)
$x_{ij}+y_{ij}+z_{ij}=1, \forall j \geq i$	(2)
$x_{ij} \geq x_{i,j+1}, \forall j \geq i$	(3)
$y_{ij} \leq y_{i,j+1}, \forall j \geq i$	(4)
$z_{ij} \leq z_{i,j+1}, \forall j \geq i$	(4)
$y_{ij}=\max\{1, 1-z_{i,j-1}\}, \forall j, i \leq M_j$	(5)
$y_{ij}+z_{ij}=1, \forall j > i+T_2$	(6)
$y_{i+1,T} \geq y_{iT}, \forall i \not\equiv N-1(\text{mod} N)$	(7)

Fig. 13: Frame Drop Strategy

A. Best GoP

As mentioned above, the value of GoP needs a tradeoff between video quality and frame dropping. The x264 encoder use the delta intermode coding, thus a larger GoP is much likely to introduce the accumulative errors, and GoP is suggested smaller than 250 frames. But how to determine the value is still an intractable question. To guide the choice of keyframe interval, we try to vary the GoP and encode many original streaming into flv format. The relationship between normalized SSIM and gop size is displayed in Fig. .

From the previous figure, we can see that value between $[20, 60]$ is always a better tradeoff.

B. Drop Strategy

1) Problem Formulation: For the constant bitrate case, assume the pace of video frame and network bandwidth are known, there exists an optimal scheduling regarding maximize audience QoE within the system constraints (bandwidth and queue timeliness length). Actually a group of pictures always compose of three kinds of frames, I/P/B frames, here for simpleness, we delete the B frame to solve the fundamental problem. The problem can be formulated by integer programming (Figure 13). Terminologies are defined in Table III. We discretize time into time stamps from 0 to T, and assume the frame with index i is generated at time i. We define x_{ij}, y_{ij}, z_{ij} as 0/1 variables to describe whether a packet is in the queue, sent or dropped.

Frame conservation constraints. Frame i is generated at time i, and after that, it is either in the queue or sent or dropped (1-2). After a packet is removed from the queue, it

Algorithm 2 Greedy Frame Enqueue Management

```
1: Input: frame, bandwidth
2: T1 := 0.9s
3: if frame is I frame then
4:     dropPFrame := False
5:     ENQUEUE(queue, frame)
6:     timespan:= timespan + 1
7: if frame is P frame then
8:     if dropPFrame or timespan ¿ T1 then
9:         DROP(frame), drop all the P frames until the next I frame
10:        dropPFrame := True
11:    else
12:        ENQUEUE(queue, frame)
13:        timespan := timespan + 1
14: timespan := timespan - SEND(bandwidth)
```

would never be enqueued (3). After a packet is sent/dropped, it is permanently sent/ dropped afterward (4-5).

Bandwidth constraints. The determination of sending strategy y_{ij}, is also an interesting and important problem. However, for simpleness, in this paper we just assume that the streamer sends as many as possible, which is a good choice. This means, at time j we just set all the possible y_{ij} to true. At any time, the number of sent frames does not exceed the network bandwidth. According to these constraints, max frame index M_j can be calculated by maximize the function $M_j = argmax(\Sigma_k y_{kj} - y_{k,j-1} \leq C_j)$. Besides the frame that can be send must be not dropped.

Timeliness constraint. A frame is "fresh" if it is sent with in "T_1". That is, a frame is either sent or drop after time T_1 of its generation (6).

Decodability constraints. The final delivered frames must be decodable; otherwise, they would be a waste of network bandwidth. I frames are always decodable. A P frame is decodable if and only if its preceding I or P frame is decodable(7).

Optimization goal. The goal of the IP model is to maximize the delivered frames. Compared with prior work [11], this IP model has timeliness and decodability in consideration, thus it is more suitable for personalized live streaming.

DP can no doubt achieve the offline optimal. But long-term bandwidth cannot be known ahead of time, so DP cannot be applied in practice. A online drop strategy is necessary.

2) Greedy Algorithm: **Algorithm Description.**Considering the encode dependency within a GoP, we propose a modified dropping algorithm(Algorithm 2) towards default obs. Different from the default dropping all the P frames in buffer, greedy algorithm optimize the corner case, where two GoP coexists in buffer. Greedy drop all the P frames until the next keyframe. This little modification performs much better.

Evaluation. We compare the performance of three algorithms, there are respectively DP, Greedy, OBS. The trace lasts for 320 seconds, and during the period both bandwidth fading and bandwidth fluctuating appears. The frame rate is 30 fps, the total number of frame equals to 9600.

TABLE IV: Dropped Frames of Three Algorithms

Algorithm	No. of dropped frames	Percentage of reduction
DP	265	17.2%
Greedy drop strategy	274	14.4%
Default OBS	320	0

TABLE V: Terminology in Adaptive Bitrate

Symbol	Type	Meaning
j	index	frame index
R_j	variable	the bitrate of frame j
N_j	variable	No. of the GoPs at time j
D_j	variable	whether frame i is dropped at time j
$Send_j$	variable	No. of frame send at time j
C_j	variable	network bandwidth at time j
T_k^j	variable	the remaining time of k GoP at time j
R_k^j	variable	the bitrate of k GoP at time j
$Drop_j$	variable	whether the drop would happen at time j
T	const	decision time

The number of dropped frames is displayed in the table IV. OBS dropped the most frames among three, and greedy reduce 14.4%, which is a notable improvement. And the gap between greedy and DP is small, less than 3%. The real-time buffer and throughput is showed in Figures 14 15, greedy algorithm has the similar behaviour with the obs, but the greedy algorithm achieves a higher minimum number of buffered frames, because greedy only drops the undecodable frames of the first GoP. Thus, the throughput when network recovers of the greedy algorithm is higher than obs. Considering both time complexity and the frame dropping, greedy is the one.

C. Adaptive Bitrate

1) Problem Formulation:

$$maximize \quad \sum R_j - \alpha \sum |R_{j+1} - R_j| - \beta \sum D_j \quad (1)$$

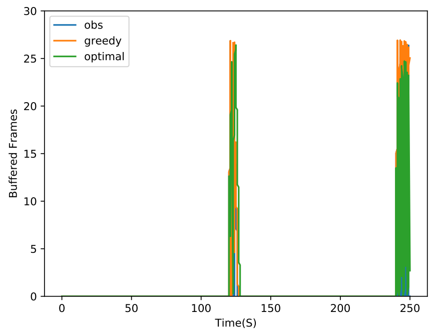


Fig. 14: Real-time buffer of three algorithms

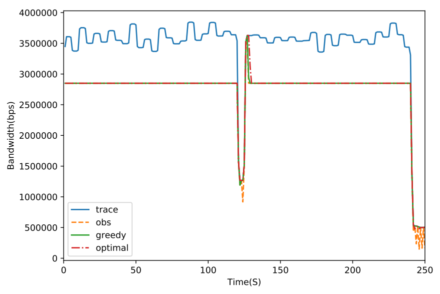


Fig. 15: Real-time throughput of four algorithms

subject to

$$R_{j+1} = R_j, \forall j \not\equiv M-1(\mathrm{mod}M) \quad (2)$$

$$S_j = argmax\sum_k R_k^j * T_k^j \le C_j, \forall j \quad (3)$$

$$F_j = sgn(\sum_{S_j+1} T_k^j - (C_j - \sum_k R_k^j * T_k^j)/R_{S_j+1}^j), \forall j \quad (4)$$

$$D_j = F_j * (C_j - \sum_{S_j} R_k^j * T_k^j)/R_{S_j+1}^j), \forall j \quad (5)$$

$$N_{j+1} = N_j - S_j - F_j + 1 - sgn(j\%M), \forall j \quad (6)$$

$$R_k^{j+1} = R_{k+S_j+F_j}^j, \forall j, k \in \{1, N_j - S_j - F_j\} \quad (7)$$

$$R_{N_j-S_j-F_j+1}^{j+1} = R_{j+1}, \forall j \equiv 0(\mathrm{mod}M) \quad (8)$$

$$T_k^{j+1} = T_{k+S_j+F_j}^j, \forall j, k \in \{1, N_j - S_j - F_j\} \quad (9)$$

$$T_{N_j-S_j-F_j}^{j+1} = T_{N_j-S_j-F_j}^{j+1} + sgn(j\%M * F_j), \forall j \quad (10)$$

$$T_{N_j-S_j-F_j+1}^{j+1} = 1, \forall j \equiv 0\mathrm{mod}M \quad (11)$$

$$(12)$$

In this section, we try to handle the long-term bandwidth fading issue. From the distribution of the bandwidth, we use the idea of adaptive bitrate similar with DASH. We use the proposed greedy strategy as the frame dropping strategy. Different from formulation upside, here how to choose the best bitrate is our point. Thus introduce a variable R_i. R_i represents the bitrate of the i frame. Problem can be formulated as follows. Variables is all defined in Table V. For variable bitrate, calculating how much frames can be send is a tricky problem, because different frames have different size. Constraint (2) requires that bitrate within one GoP must keep the same; (3) calculates the most number of GoPs can be send, constraint (4) judges whether the remaining time after sending exceeds the buffer limit. Equation (5) is the number of dropped frames. Constraints (6) – (11) show the state transition of the bitrate and time of several GoPs.

Offline optimal is hard to calculate. Guess for each GoP, the broadcaster can choose one from total M candidates. For a T GoP decision, the computation complexity equals to M^T, a exponential complexity.

2) Effective Solution: **Algorithm Description.** A exponential complexity issue is hard to calculate in limited time.

Algorithm 3 Greedy Video Adaptation Workflow

```
1: Initialize
2: for j=1 to T do
3:     $C_j := ThroughpurPred(C_{[\tau,j-1]})$
4:     $R_j := FindClosestBr(C_j)$
5:     $D_j := f(C_j, R_j)$
```

Besides, the offline optimal is on the basis of given prefect knowledge of future bandwidth. Such long-term bandwidth prediction is inaccurate. A intuitive idea is to change the bitrate following the bandwidth. At time j, the broadcaster carries out the following two key steps, as shown in Algorithm 3.

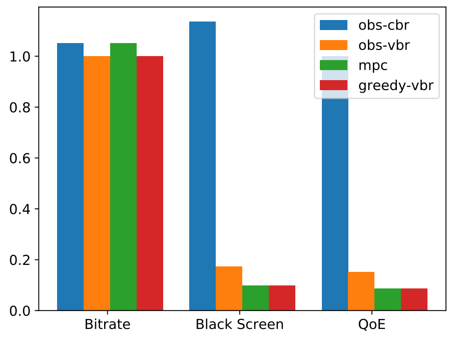
1. Bandwidth estimate. According to Festive and MPC, harmonic mean is a useful method of estimating the future bandwidth. Besides, proposing a prediction mechanism is not the focus.

2. Bitrate choose. Avoid from frequent frame dropping, an appropriate bitrate is essential. Given the future bandwidth C_j, an heuristic choice is to choose the highest available bitrate lower than C_j.

Evaluation. Buffer-based algorithm preforms excellently in DASH. But in live streaming scenario, the buffer only lasts for $0.7 - 0.9$ seconds, buffer-based is hard to apply and can only make little difference. But the bitrate-based algorithm still can use. Inspired from refXiaoqi Yin, the state-of-art MPC is used as the baseline.

We compared four algorithm together. First is the obs default algorithm, constant bitrate, and the default drop strategy. Second, change the constant bitrate into the greedy algorithm. Third, compare the greedy drop strategy plus the RHC. And the last one, our own greedy bitrate chosen algorithm plus the greedy drop strategy. Specific comparison results are shown in Figures 16a.

These algorithms reach almost the same average bitrate, with little difference. Because for all of these algorithms, the real-time bitrate waves around the average bandwidth. The first CBR algorithm drops the most frames, more than 5 times when compared with other VBR algorithms. Introducing the idea of VBR reduces the dropped frames to a acceptable



(a) Normalized bitrate, black screen and QoE for four algorithms

level, at most 5 seconds. QoE equals to the weighted sum of three key factors. In our case, frame dropping takes the most important roles, the value of QoE almost follows the frame dropping. MPC and our proposed greedy algorithm perform almost the same in QoE and frame dropping. Default OBS always has a poorer qoe than other three algorithms. The following cdf shows the specific distribution of all the three indexes, similar with the average pictures.

VI. Related Work

VII. Conclusion and Future Work

In the future, we would first implement and test individual design methods in Section **??**, including adaptive GOP selection and online frame drop strategy. In our expectation, the final solution would be a combination of these design methods. For example, increase queue threshold, selectively drop frames if the threshold is reached, and when the network recovers, pick a recent frame, recompute the GOP, and send the timely frames to the network.

References

[1] https://live.fb.com/.
[2] https://www.douyu.com/.
[3] https://www.pscp.tv/.
[4] https://www.twitch.tv/.
[5] http://www.iqiyi.com/.
[6] https://en.wikipedia.org/wiki/Meerkat_(app).
[7] J. Huang, C. Krasic, and J. Walpole. Adaptive live video streaming by priority drop. In *AVSS'03 Proceedings of the IEEE Conference on Advanced Video and Signal Based Surveillance*, 2003.
[8] C. Krasic, J. Walpole, and W.-c. Feng. Quality-adaptive media streaming by priority drop. In *Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, pages 112–121. ACM, 2003.
[9] B. Seo, W. Cui, and R. Zimmermann. An experimental study of video uploading from mobile devices with http streaming. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 215–225. ACM, 2012.
[10] M. Siekkinen, E. Masala, and T. Kämäräinen. A first look at quality of mobile live streaming experience: the case of periscope. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, pages 477–483. ACM, 2016.
[11] S. K. Singh, H. W. Leong, and S. N. Chakravarty. A dynamic-priority based approach to streaming video over cellular network. In *Computer Communications and Networks, 2004. ICCCN 2004. Proceedings. 13th International Conference on*, pages 281–286. IEEE, 2004.
[12] J. C. Tang, G. Venolia, and K. M. Inkpen. 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*, pages 4770–4780. ACM, 2016.
[13] B. Wang, X. Zhang, G. Wang, H. Zheng, and B. Y. Zhao. Anatomy of a personalized livestreaming system. In *Proceedings of the 2016 ACM on Internet Measurement Conference*, pages 485–498. ACM, 2016.
[14] S. Wilk, R. Zimmermann, and W. Effelsberg. Leveraging transitions for the upload of user-generated mobile video. In *Proceedings of the 8th International Workshop on Mobile Video*, page 5. ACM, 2016.
[15] C. Zhang and J. Liu. 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*, pages 55–60. ACM, 2015.

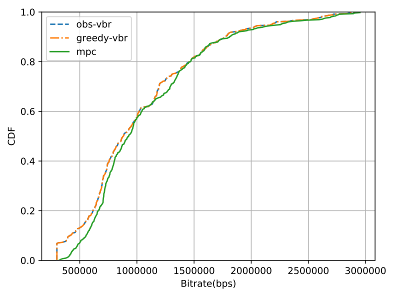


Fig. 17: Bitrate Cdf of four algorithms

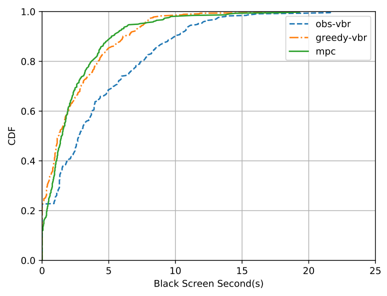


Fig. 18: Black Screen Seconds Cdf of four algorithms

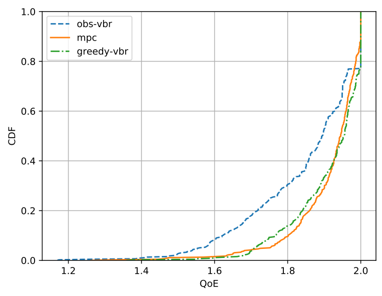


Fig. 19: Average Normalized QoE Cdf of four algorithms