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 [2], iQIYI [8]), sharing richer experience (e.g., Facebook Live [3], Periscope [6]), and broadcasting online gaming and sports events (e.g., Twitch [7], Douyu [5]).

While recent work on personalized live streaming has insofar focused on its traffic pattern (e.g., [20], [16]) and video distribution architecture (e.g., [14], [17]), 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-

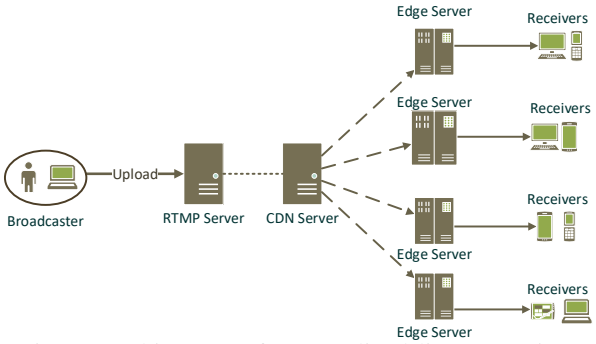


Fig. 1: Architecture of personalized live streaming.

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., [10], [11], [15]). 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 [13], and adaptively switching between them [19]. 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. The demo comprises of two

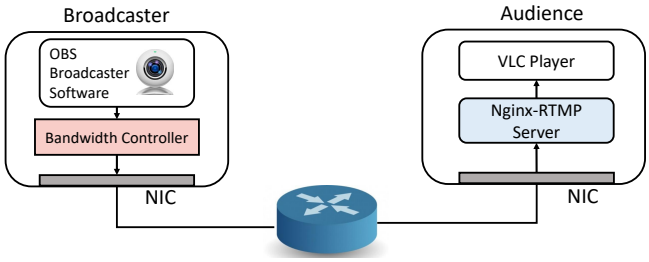
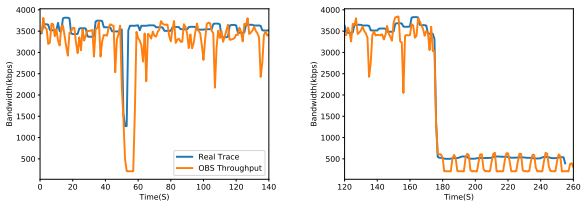


Fig. 2: Experiment setup

modules, sender and receiver, which are connected by a switch in the middle. Servers both have 2 CPU cores and 6GB memory, and are equipped with 100Mbps NICs. OBS studio[4] is one popular broadcast software and is used to stream videos to the audience side over RTMP protocol in sender's side. We use both the tc module of linux and dummynet[1] to control the real-time upload bandwidth of sender. The broadcaster server(receiver) is built on nginx-rtmp module. On the audience server (receiver), VLC player is used to play the rtmp streaming.

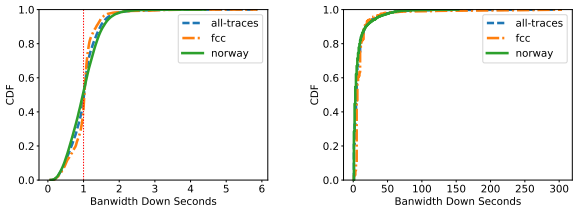


(a) Measured throughput of $0-140$s (b) Measured throughput of $120-260$s

Fig. 3: 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 records the real-time 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 slot. We then control the network bandwidth on the sender side according to the per-second profile. The average bandwidth is up to 3000kps, we stream video at a bitrate of 3000kbps via OBS and capture actual video packet trace using tcpdump in the receiver's side. And the result is shown in Figure 3. The trace lasts for 320s, such a long time that we break the trace into two parts.

In the figure 3a, the actual throughput follows the trace closely. However, at 50s, the network bandwidth falls below the bitrate and the situation lasts for 2 seconds, while the actual throughput degrades to almost zero from 50s to 58s. 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 3b. Bandwidth is enough during $0-180s$, but after $180s$, the available bandwidth drops dramatically and lasts for $80s$, endless frames drop in this period. In this challenging network environment, the default OBS insists previous bitrate and obviously the strategy is not good.



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

Fig. 4: Trace analysis: Bandwidth down in wireless network

Network Conditions To know how often the bandwidth failure occurs, first we want to know the bandwidth distribution of real world. Two real-world dataset, FCC dataset[**?**] and HSDPA dataset[**?**], is combined to calculate the bandwidth failure ratio. Each trace lasts for 320s, and the total dataset lasts for 30 hours. For each trace, referring the average bandwidth as the unit, we normalize the trace and draw the cdf(Figure. 4a). Almost 50% of traces are under the average throughput, which means for a 10 second trace, about 5 second the bandwidth is lower than the average. About 20% of the traces are at most half of the average. The figure indicates that in real-time network, bandwidth fluctuation frequently occurs. To further explain how often long-term bandwidth fluctuation happens, we draw a picture of network failure time distribution, Figure 4b. Network failure time is calculated by counting the lasting 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.

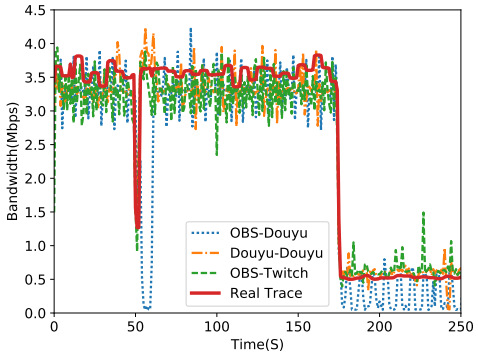


Fig. 11: Bandwidth Control

Experiments on several commercial platforms. We further repeat the experiment in different commercial platforms

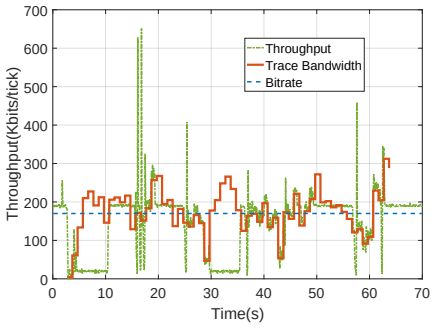


Fig. 5: Throughput, OBS to Douyu server

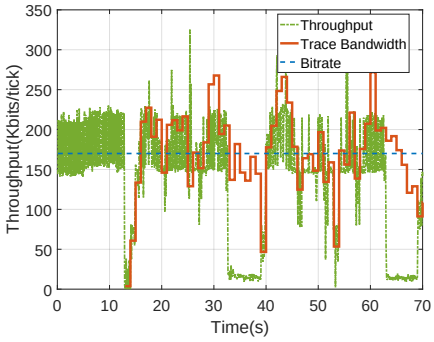


Fig. 6: Throughput, Douyu broadcaster to Douyu server

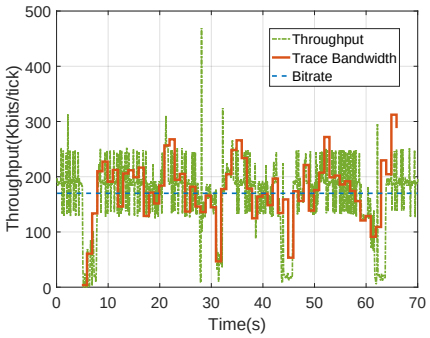


Fig. 7: Throuhput, OBS to Twitch server

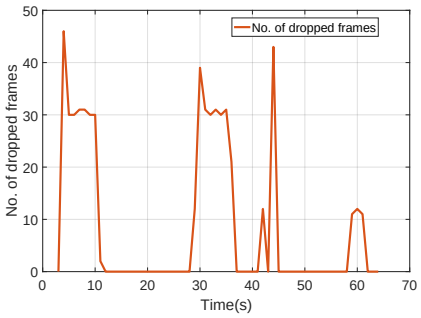


Fig. 8: Dropped frames, OBS to Douyu server

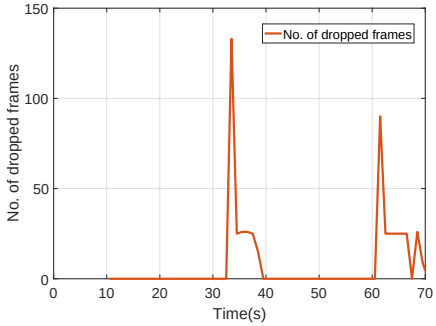


Fig. 9: Dropped frames, Douyu broadcaster to Douyu server

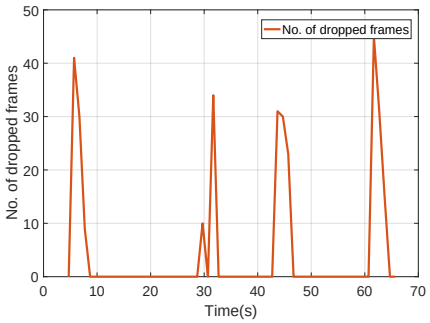


Fig. 10: Dropped frames, OBS to Twitch server

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%

and settings to find whether the same issues exists. We test three different situations, including OBS pushing video to Douyu server, Douyu broadcaster to Douyu server, and OBS to twitch server. In three experiments, the bitrate is choose lower than the average bandwidth, 1700kbps. Tcpdump is used to record the real-time throughput, we aggregate the data size in one tick(0.1s) together and draw a picture. Figure 5, 6, and 7 show the throughput of the three experiments respectively, Figure 8, 9, 10 is the corresponding number of dropped frames. The first three lines in Table I show the total 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). For these three time period, the frame dropping keeps a high value. We also find out that the cascading effect is not related to the instantaneously available bandwidth. For example, in Figure 5, a dramatic bandwidth drop at 30s causes the cascading frame drop; while in Figure 6, 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: 5+s 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.

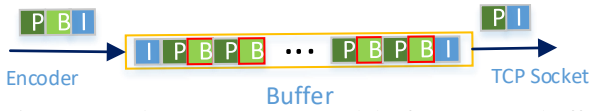


Fig. 12: Producer-consumer model of streamer's buffer

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. 13: 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 13 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 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.

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)
```

C. Design Space Insight

To meet the delay constraints, there are two kinds of solution. One is to limit the timeliness of the sender buffer, strictly restrain each frame to meet the time requirements; the other maybe controlled by scheduling to maintain the average of delay at the target value, this is . 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. **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.

TABLE II: 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	key 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)
$y_{i+1,T} \geq y_{iT}, \forall i \not\equiv N - 1 (\mathrm{mod} N)$	(7)

Fig. 14: Frame Drop Strategy

Improve the frame drop strategy. The default strategy in OBS is dropping all P/B frames in buffer when exceeding a threshold. It's some reasonable because if dropping the earlier frames, the following frames cannot be decodable; and if dropping the latest several frames, the earlier frames still exist in buffer, the timeliness will be violated. But 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 lies in the complexity of the frame dependency. A brute force solution is impossible because its time complexity.

Adaptive bitrate. Network failure occurs frequently. Conclusions from figure 4 validate the fact. Previous broadcasters only use constant bitrate(CBR) or ABR, which means the actual bitrate varies among the target value, at most 20% lower or higher 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 certain while. One possible solution is similar with DASH in VOD scenario, applying adaptive bitrate in broadcaster's side. Introducing bitrate adaptation maybe dramatically cut down the frame dropping.

IV. Solution

According to the three principles, better drop strategy and practical video adaptation is discussed in this section.

A. 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 study the fundamental problem. The problem can be formulated by integer programming (Figure 14). Terminologies are defined in Table II. 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 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, the decision of y_{ij}, is also an interesting and important problem. However, for simpleness, in this paper we just assume that the broadcaster sends as many as possible, which is a good choice. This means, at time j we send out all the possible frames and set the corresponding y_{ij} to true. At any time, the number of sent frames should not exceed the available network bandwidth. According to these constraints, the max frame index M_j that can be send, is calculated by maximize the function.

$$M_j = argmax \Sigma_k (y_{kj} - y_{k,j-1})(1 - z_{k,j-1}) \leq C_j \quad (1)$$

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 [15], 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. An online drop strategy is necessary.

2) Greedy Algorithm: **Algorithm Description.** Considering the encode dependency within a GoP, we propose a modified dropping algorithm, GreedyDrop Algorithm 2 towards default OBS. Different from the default dropping all the P frames in buffer, greedy algorithm optimizes one more case, where two or more GoPs coexist in buffer. Greedy drops all the P frames until the next keyframe such that the latest GoP can be reserved and avoid frame dropping at least one GoP. The little modification performs much better.

Algorithm 2 GreedyDrop Algorithm

```
Input: frame, bandwidth
T1 := 0.9s
if frame is I frame then
    dropPFrame := False
    ENQUEUE(queue, frame)
    timespan:= timespan + 1
if frame is P frame then
    if dropPFrame or timespan ¿ T1 then
        DROP(frame), drop all the P frames until the next I frame
        dropPFrame := True
    else
        ENQUEUE(queue, frame)
        timespan := timespan + 1
timespan := timespan - TIME-SEND(bandwidth)
```

TABLE III: 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
S_j	variable	No. of GoPs send at time j
C_j	variable	network bandwidth at time j
T_k^j	variable	the remaining time of k-th GoP at time j
R_k^j	variable	the bitrate of k-th GoP at time j
$Drop_j$	variable	whether the drop would happen at time j
T	const	decision time
T_1	const	max time when a frame keeps "fresh"

B. Adaptive Bitrate

1) Problem Formulation:

$$Max \quad \sum R_j - \alpha \sum |R_{j+1} - R_j| - \beta \sum D_j \quad (2)$$

subject to

$$R_{j+1} = R_j, \forall mod(j, M) \not\equiv M - 1 \quad (3)$$

$$S_j = argmax \sum_k R_k^j * T_k^j \leq C_j, \forall j \quad (4)$$

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

$$F_j = sgn(\sum_{S_j+1} T_k^j - Rest_j - T_1), \forall j \quad (6)$$

$$D_j = F_j * (T_{S_j+1}^j - Rest_j), \forall j \quad (7)$$

$$N_{j+1} = N_j - S_j - F_j + 1 - sgn(mod(j, M)), \forall j \quad (8)$$

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

$$R_{N_j-S_j-F_j+1}^{j+1} = R_{j+1}, \forall mod(j, M) \equiv 0 \quad (10)$$

$$T_k^{j+1} = T_{k+S_j+F_j}^j, \forall j, k \in \{1, N_j - S_j - F_j\} \quad (11)$$

$$T_{N_j-S_j-F_j}^{j+1} = T_{N_j-S_j-F_j}^{j+1} - D_j - Rest_j, \forall j \quad (12)$$

$$T_{N_j-S_j-F_j+1}^{j+1} = 1, \forall mod(j, M) \equiv 0 \quad (13)$$

$$(14)$$

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. Different from the former issue, 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. For variable bitrate, calculating how much frames can be send is a tricky problem, because different frames have different size.

Problem can be formulated as follows 2, Variables is all defined in Table III. Variables and are the utility parameter of bitrate switch and frame dropping. sgn is the sign function, when the variable greater than zero, it equals 1; otherwise equals zero. mod is the operation of taking remainder.

Bitrate Constraint Constraint 3 requires that bitrate within one GoP must keep the same.

Bandwidth Constraint Equation 4 calculates the most number of GoPs can be send within the limited bandwidth.

Timeliness Constraint 6 judges whether the remaining time after sending exceeds the buffer limit and 7 give the number of dropped frames in time j.

State Transition Constraints 9, 10, 11, 12, 13 show the state transition of the bitrate and remaining time of several GoPs in buffer. Equations 8 describes the number of GoPs in the next time slot $j+1$, the last two items $1-sgn(jmodM)$ represents whether the $j-th$ frame is the keyframe.

Offline optimal is hard to calculate. Guess for each GoP, the broadcaster can choose one from total M bitrate 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. 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

Algorithm 3 Greedy VBR algorithm

Initialize Rest=0, Send=0, Drop=0, α
for j=1 to T **do**
    according to the history bandwidth $[C_{j-tau}, C_{j-1}]$, use harmonic mean to estimate C_j
    choose the closest bitrate R_j to $(C_j - rest)/\alpha$
    send frames $Send$ in buffer within the bandwidth limit
    judge whether to drop extra frames $Drop$
    calculate the remaining data size in buffer $Rest = Rest + R_j - Send - Drop$

bitrate following the bandwidth. Besides, the remaining data size in buffer may also be helpful. At time j, the broadcaster carries out the following two key steps, as shown in Greedy VBR (GVBR) 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 nots our focus. With more accurate bandwidth estimate, our method will be better.

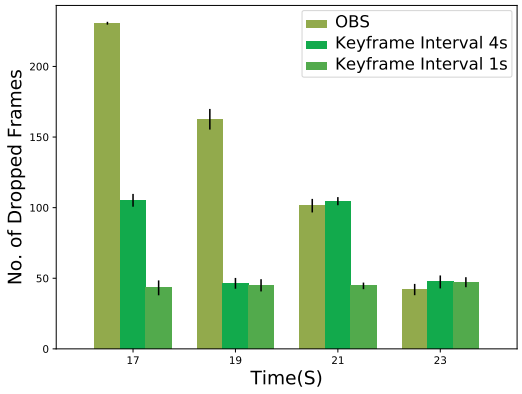
2. Bitrate choose. Avoid from frequent frame dropping, an appropriate bitrate is essential. Given the future bandwidth C_j and the data size in buffer $Rest$, an heuristic choice is to choose the highest available bitrate lower than $(C_j - Rest)/\alpha$.

V. Evaluation

A. Best GoP

The previous section indicates that reduce the keyframe interval may cut down the frame dropping. And in this section, we evaluate the method: reducing keyframe interval.

Implementation and experiment setup. We design control experiments, where we control the outbound throughput of broadcaster to a certain level, and introduces a 2-second interruption. We record the number of frame drop as metrics to evaluate these methods. The frame rate in this paper usually equals to 30.



(a) Frame drop with varying I frame interval

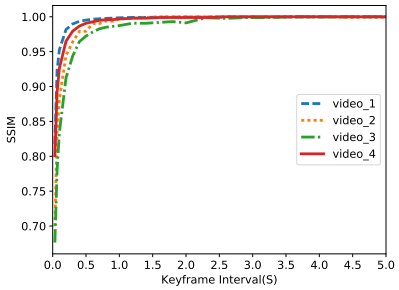


Fig. 15: SSIM for different GoP values

Varying key frame interval. The default I frame interval of OBS is 8 seconds, and we adjust it to be 4s and 2s in experiments. The frame drop are shown in Figure 15a. 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, for default setting, 8 second keyframe interval, when interruption starts at 17s, 19s, 21s, and 23s, the number of frame drop is 238, 164, 105, and 48. Also, the number of frame drop appears to have the same period with the keyframe interval (e.g., when keyframe interval is 4s, the number of frame drop is 102, 47, 103, and 48 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 238, to 102, and 46 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.

Video quality and GoP As mentioned above, the value of GoP needs a tradeoff between video quality and frame dropping. To guide the choice of keyframe interval, we try to vary the GoP and encode many original streaming using x264 encoder[9]. A truth is that 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 specific value is still intractable. We record SSIM as metrics. SSIM, the abbreviation of structural similarity, is a method for predicting quality of video, and is used for measuring the similarity of two images[18]. The video dataset we use contains SD content, HD content, gaming, 4k content in

TABLE IV: Dropped Frames of Three Algorithms

Algorithm	No. of dropped frames	Percentage
DP	265	80%
Greedy drop strategy	274	85.6%
Default OBS	320	100%

variety[**?**]. The relationship between normalized SSIM and gop size is displayed in Fig 15.

We pick four from many videos to represent the result. From the figure, we can see when the GoP size is larger than 20, the SSIM keeps almost the same, with little changing.

Combined the previous two experiments, a GoP size larger than 20 can achieve better video quality; and a smaller gop size will reduce the frame dropping. We can see that value between $[20, 60]$ may be almost the best choice for keyframe interval.

B. Greedy Drop Strategy

To measure the performance, we compare the performance with two algorithms, there are respectively Oracle, OBS default. Oracle is the brute-force search to calculate the optimal solution, which has an exponential time complexity. We pick out one part from the dataset, and the trace lasts for 30 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 900.

The number of dropped frames is displayed in the table IV. OBS dropped the most frames among three, and Greedy-Drop reduce 15%, which is a notable improvement. And the gap between GreedyDrop and Oracle is small, less than 5%. The real-time frame drop and throughput is showed in Figures 16 17, GreedyDrop has the similar drop behaviour with the obs, but GreedyDrop drops less frame in 20-30s. Because Greedy only drops the undecodable frames of the first GoP. And thus, the throughput when network recovers of the greedy algorithm is higher than obs, which can be seen in 12-14s. Optimal will save more frame before the network recovers, and keep a high bandwidth. But for an online algorithm, it's much diffcult. Considering both time complexity and performance, GreedyDrop is a good choice.

C. Greedy Adaptive Bitrate

We compared GVBR algorihtm towards three algorithms which work excellently in VOD bitrate adaptation:

- OBS: A simple video adaptation method, each time choose the bitrate exactly lower than the estimated bandwidth. Besides, the default drop strategy in obs. Harmonic mean is used for bandwidth estimation.
- MPC: use buffer state(number and size of frames, and type of frames, I/P/B) and bandwidth predictions to calculate the optimal bitrate operation in future several time slots; and use the first bitrate choice in next time slot.
- Robust-MPC: use the approach similar with MPC, and correct the estimated bandwidth by considering the prediction error in past several time slots. New estimated bandwidth equals to the original estimated bandwidth divide the prediction error.

Detailed comparison results are shown in Figures 18 and 19. In these two figures, the red real line is one real-world trace. Without predication error, MPC prefers to choose more higher bitrate than Robust-MPC. Besides, these two MPC algorihtm always shake around the real bandwidth. In both cases, the MPC and Roubust-MPC switch more bitrate than GVBR. Because GVBR tends to choose the lower bitrate than the throughput, and when comes the small bandwidth fluctuation, GVBR is less likely to shake. But MPC struggles to achieve the optimal utility, and when the bandwidth increases a little, MPC has the potential to choose a higher bitrate to maximize the first item in 2.

A massive simulation is displayed in Figure 20. We use the combined dataset, FCC and HSDPA to evaluate GVBR algorithm. Normalized qoe is calculated in the figure. Among all, MPC has the max bitrate qoe, because the bandwidth estimation is aggressive and MPC has the potential to choose higher bitrate. Three others reach almost the same average bitrate, with little difference, but GVBR is a little higher. With higher bitrate, MPC also drops the most frames, and the time of play failure is longest. GVBR reduces the play failure to a small value, 50% reduction compared with Robust-MPC. With higher bitrate and lower play failure, GVBR definitely preforms the best, with the highest QoE.

Cdf figures about each metric is as follows, 21, 22, 23. In 21, GVBR lies in the right of Robust-MPC and OBS, with a higher bitrate. 98% of the paly failure is less than 5s in GVBR, about 40% play fluently with no failure. Only 2% of GVBR receive poor qoe, the rest 98% has a high qoe lies in [0.8,1].

As all, GVBR achieve a higher bitrate, and at the same time reduce the play failure to little.

VI. Related Work

Recently abundant works focus on seamlessly bitrate adaptation of DASH, which is called dynamic adaptive streaming over http. All the VBR algorithm is DASH can be classified into several categories: rate-based and buffer-based and a combination of both two. Rate-based methods often pick the highest available bitrate lower than the estimated bandwidth; buffer-based algorithms choose the bitrate according to the buffer level: at a high buffer level, it prefers to choose a higher bitrate; and at a low buffer level, a lower bitrate instead. Control theory is also applied into the bitrate adaptation, which is called MPC. MPC forecasts the future network bandwidth of several chunks, and finds the optimal solution during these periods, then applies the first choice. MPC preforms better than all others. There are also some papers about using DASH in live streaming scenario. [12] includes the idea of adaptive streaming in live streaming, but its focus is how to implement in a massive scale and the difficulty mainly lies in the resource management. The difference

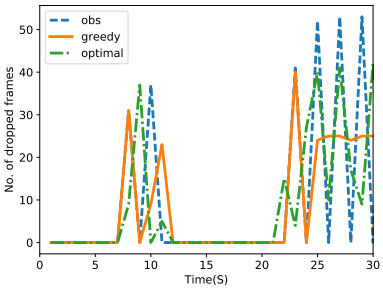


Fig. 16: Real-time buffer of three algorithms

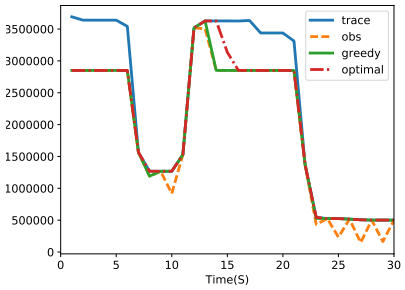


Fig. 17: Real-time throughput of four algorithms

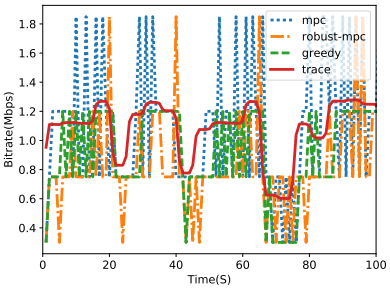


Fig. 18: Throughput of FCC dataset

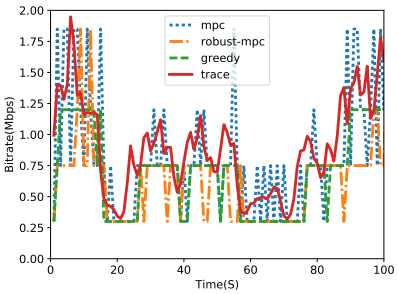


Fig. 19: Throughput of HSDPA dataset

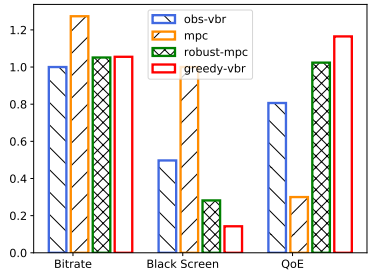


Fig. 20: Normalized bitrate, black screen and QoE for four algorithms

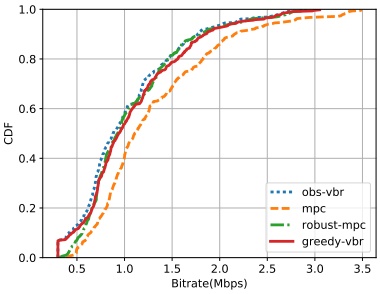


Fig. 21: Bitrate Cdf of four algorithms

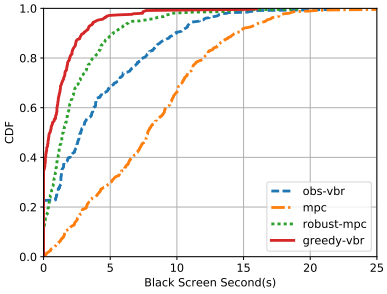


Fig. 22: Black Screen Seconds Cdf of four algorithms

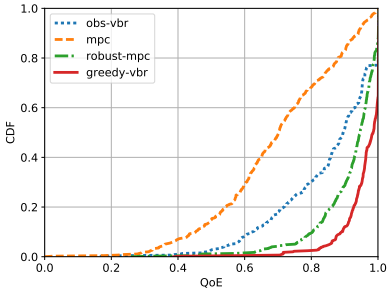


Fig. 23: Average Normalized QoE Cdf of four algorithms

between DASH and video adaptation in live streaming is the following aspects. One, the time granularity, in DASH, the time slot lasts for 2-10 seconds, but in live streaming, the time slot lasts for less than 2 seconds; two, the buffer size, DASH is mainly used in VOD, the buffer usually equals to dozens of seconds, but in live streaming, the buffer almost is less than 1 second. And most important, all of these are at the viewer's side, researches about the broadcaster's side is little.

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] http://info.iet.unipi.it/ luigi/dummynet/.
[2] https://en.wikipedia.org/wiki/meerkat(app).
[3] https://live.fb.com/.
[4] https://obsproject.com/.
[5] https://www.douyu.com/.
[6] https://www.pscp.tv/.
[7] https://www.twitch.tv/.
[8] http://www.iqiyi.com/.
[9] http://www.videolan.org/developers/x264.html.

[10] 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.

[11] 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.

[12] K. Pires and G. Simon. Dash in twitch: Adaptive bitrate streaming in live game streaming platforms. In *Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming*, pages 13–18. ACM, 2014.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[19] 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.

[20] 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.