Enabling High Quality Real-Time Communications with Adaptive Frame-Rate

Zili Meng[1,2], Tingfeng Wang[1,2,3], Yixin Shen[1], Bo Wang[1,4], Mingwei Xu[1,4],
Rui Han[2], Honghao Liu[2], Venkat Arun[5], Hongxin Hu[6], Xue Wei[2]
[1]*Tsinghua University,* [2]*Tencent Inc.,* [3]*Beijing University of Posts and Telecommunications,*
[4]*Zhongguancun Laboratory,* [5]*Massachusetts Institute of Technology,* [6]*University at Buffalo, SUNY*

Abstract

Emerging high-quality real-time communication (RTC) applications stream ultra-high-definition (UHD) videos with a high frame rate (HFR). They use edge computing, which enables high bandwidth and low latency streaming. Our measurements, from the cloud gaming platform of one of the largest gaming companies, show that, in this setting, the queue at the client-side decoder is often the cause of high latency that hurts the user's experience. We, therefore, propose an Adaptive Frame Rate (AFR) controller that helps achieve ultra-low latency by adaptively coordinating the frame rate with fluctuating network conditions and decoder capacity. AFR's design addresses two key challenges: (1) queue measurements do not provide timely feedback for the control loop; and (2) multiple factors control the decoder queue, and different actions must be taken depending on why the queue accumulates. Both trace-driven simulations and large-scale deployments in the wild demonstrate that AFR can reduce the tail queuing delay by up to 7.4× and the stuttering events measured by end-to-end delay by 34% on average. AFR has been deployed in production in our cloud gaming service for over one year.

1 Introduction

Emerging network technologies like 5G have gotten both academia and industry excited about high-quality real-time communication (RTC) applications with ultra-high definition (UHD), high frame rate (HFR), and reduced delays. Examples include cloud gaming [41, 82], virtual reality [37, 88, 63] and 4K video conferencing [40, 49]. Some high-quality RTC services have already been deployed in production (e.g., cloud gaming from Google [3], Microsoft [1], Nvidia [5]). For example, the market share of cloud gaming reached one billion dollars in 2020, with an expected growth rate of 30% [19].

To achieve a satisfactory user experience, those applications need to stream with high resolution, high frame rate, and a low delay (§2). For example, cloud gaming services deliver content with a resolution of ≥1080p [3] and frame-rate of 60fps [61], while requiring a tail end-to-end delay of less than 100ms [43]. Streaming like this significantly improves users' experience and enables new applications.

This paper argues that, in addition to modulating bitrate to match network capacity, a high-quality RTC system must regulate the queuing at the decoder queue. For traditional standard quality RTC, the time required to decode a frame is much shorter than the interarrival time of frames. Thus, the decoder queue is not a bottleneck and a traditional RTC service only needs to adjust the bitrate to match the network bandwidth. However, in high-quality RTC, the high frame rate reduces the time between the arrival of frames at the client, while the high resolution increases the decoding delay for each frame. At the decoder queue, the frame arrival rate frequently exceeds the departure rate, leading to a long queue, as shown in Figure 1. The video delivery is required to not only adapt the bit-rate to the network bandwidth but also coordinate with the decoder queue capacity. From measurements of our production cloud gaming service, Tencent Start [4], we find that video delivery without coordinating the queue capacity could introduce a non-negligible queuing delay at the client-side decoder queue. Moreover, such a queuing delay accounts for a large proportion of delayed frames in satisfying the much tighter delay requirement of high-quality RTC, especially when the network delay has been reduced with recent infrastructure developments (e.g., edge computing [57]). According to our measurements, among all frames with a total round-trip delay of >100ms, 57% of them have been delayed at the decoder queue for >50ms (§3.1). Our survey finds that the future demands of UHD and HFR video will further exacerbate the problem, even with the evolution of decoding hardware (§3.1). Therefore, for high-quality RTC, to reduce the end-to-end delay, it is essential to reduce the queuing delay at the decoder.

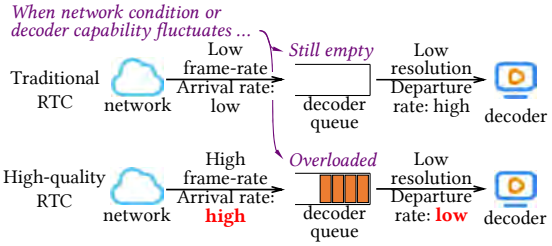


Figure 1: Comparison of the decoder queue between traditional and high-quality RTC applications. Due to the high frame rate and resolution, when network condition or decoder capability fluctuates, high-quality RTC applications may overload decoder queues, leading to high delay at the tail.

Not all interventions are effective at regulating the queuing at the decoder queue (§3.2). For instance, decoding delay is not affected much by bitrate. It is affected by resolution, but adjusting the resolution requires the client to request a new key frame. This consumes bandwidth and incurs several extra frame intervals of delay. Discarding a frame at the client also requires a new key frame, which incurs the same cost. Hence, we introduce an *adaptive frame-rate (AFR)* controller, which controls the frame rate at the *encoder*. Reducing frame rate gives the decoder more time to process frames. Hence, it is effective at reducing the queue length. Further, edge streaming services offer short RTTs, which means the control loop to adjust the encoder's frame rate is short.

Note, there have been previous efforts to adapt the frame-rate

(e.g., CU-SeeMe [38] decades ago). However, the development of decoding hardware had made it redundant in the recent decade, and traditional RTC in the recent decade is mostly bottlenecked in the network. In this paper, we show how high-quality RTC, with UHD resolution, HFR, and stringent delay requirements, has changed this. We further improve upon these proposals in two ways. First, existing control mechanisms are based on delay or queue length [60, 34, 77], which are slow to react since they need to wait for the queue to build up. AFR instead relies on the arrival and service processes in addition to the queue length to adjust the frame rate. Second, not all increases in decode queuing delay need to reduce the frame rate. For instance, when queuing delay increases due to a transient burst of arriving packets. Hence, AFR uses two control loops that adjust the frame rate at different time scales.

We implement the AFR controller on both simulators and the production of the cloud gaming service from Tencent Start [4]. Trace-driven simulations and deployments in the wild demonstrate that AFR could effectively reduce the tail queuing delay by up to 7.4×, and consequently reduce the ratio of frame stutters measured by total delay by up to 2.2× (§6.1 and §6.5) with negligible overhead. AFR has been deployed on Tencent Start since February 2021, serving millions of sessions. We will release the collected traces and the simulation code.

We make the following contributions:

- We carry out a month-long measurement campaign to motivate the significance of controlling queuing delay at the decoder queue, and identify the unique challenges from high-quality RTC with stringent delay requirements (§3).
- We design a hierarchical frame-rate controller, AFR, to control the decoder queue towards an ultra-short delay under different scenarios for high-quality RTC (§4).
- We evaluate AFR with both trace-driven simulations and large-scale deployments in production in the wild (§5). Our evaluation shows that both queuing delay and total end-to-end delay could be significantly improved (§6). AFR has been used in deployment for over one year.

2 Background: High-Quality RTC

High-quality RTC applications are attracting attention from the industry and academia. A series of high-quality RTC products have been released recently, including cloud gaming [3, 1, 5], virtual reality (VR) [12, 11, 6], and 4K videoconferencing [8]. For example, by generating high-quality content and streaming to the user via Internet, users can enjoy better video quality with low-cost devices. Specifically, the high-quality RTC has the following features standing out from traditional RTC applications:

- *High frame-rate*. Traditional RTC usually delivers content with a low frame rate (LFR) of 24fps [9]. However, high-quality RTC requires a frame rate of up to 60fps, some of which even require a frame-rate of 240fps [73].
- *High resolution*. Most existing RTC applications are delivered at SD resolutions by default (e.g., 360p for Google Meet [7]). In contrast, high-quality RTC applications require a resolution of 1080p to 4K or higher [62].
- *Stringent delay requirement*. Furthermore, high-quality RTC applications also have stringent latency requirements. For example, videoconferencing requires a round-trip interaction delay of 150ms [9] and gaming for 100ms [43].

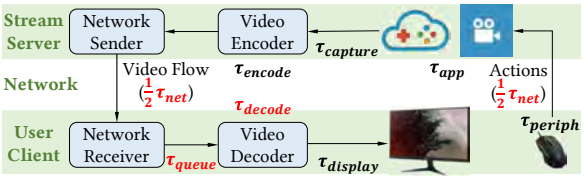


Figure 2: A general delivery pipeline of RTC services. We highlight the major contributing components in the tail end-to-end delay of high-quality RTC according to our measurements in red.

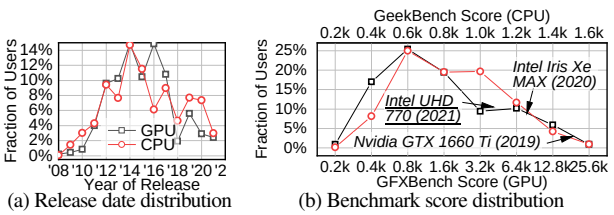
Existing delivery pipeline. To better understand the bottleneck of high-quality RTC, we present the key components of the existing RTC delivery pipeline in Figure 2. First, the video encoder captures the contents generated from video sources (e.g., gaming applications [23, 57]) and encodes them into video frames. Then, encoded frames are sent over the network from the streaming server to user clients. After that, on the client side, upon receiving new frames from the network, the decoder will decode those frames. Finally, decoded video frames will be displayed on users' displays.

Optimization goal: low tail delay. With the intelligence from each community, the delay of each component has been intensively optimized in recent research efforts. To reduce the network delay, existing providers either deploy stream servers at the edge [57, 74], introduce low-latency congestion controllers [16, 25], or suggest users use wired connections. For example, recent measurements unveil that cloud gaming services could deliver the RTC streams with an average round-trip network delay of 20ms [57, 26]. Similarly, streaming encoders are optimized for low latency to satisfy the stringent delay requirements in high-quality RTC services [58, 69, 34].

Meanwhile, optimizing the *tail* performance is also critical for user's experience for high-quality RTC [56]. The increase in tail delay will result in frame stuttering or freezing, degrading the user's experience. Quality of experience assessment frameworks in video streaming usually individually calculate the stuttering time as a penalty to the user's experience [33, 80]. Considering the high frame rate of high-quality RTC, further tails of 99th or 99.9th percentiles need to be focused on. For example, at the frame rate of 60fps, even the 99.9th percentile delay could happen every 16 seconds. Especially for applications such as cloud gaming, such a delay might lead to the loss of the game (e.g., stalls when the gamer is discovered by the opponent in a shooting game) [67, 43]. Therefore, it is essential to control the tail delay and reduce frame stutters for high-quality RTC.

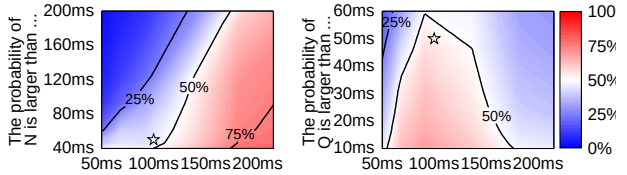
3 Motivations and Challenges

In this section, we first explain the formulation of drastic queuing delay in high-quality RTC (§3.1). We then present our thinking over the design choice of adjusting frame rate (§3.2). We



(a) Release date distribution (b) Benchmark score distribution

Figure 3: Release year and benchmark score distribution of user devices in production. We use the single-core score in GeekBench [15] for the CPU benchmark and Aztec Ruins Normal Tier score in GFXBench [13] for the GPU benchmark.



(a) Network delay. (b) Queuing delay.

Figure 4: While network delay should usually be blamed when the total delay is above 200ms, queuing delay plays a dominant role among all frames with a total delay of more than 100ms. The color indicates the conditional probability $P(X > X_{th} | T > T_{th})$ for $X \in \{N, Q\}$. Stars denote X_{th}=50ms, T_{th}=100ms.

further analyze the unique challenges of effectively achieving an ultra-short queue (§3.3).

3.1 Motivation: Drastic Queuing Delay

Observation: decoder queuing delay is a critical contributor to the total delay at the tail. We profile the delay of each frame at each stage in the delivery pipeline in Figure 2. We measure the Tencent Start cloud gaming service for a month in 2021, containing tens of thousands of users, with thousands of different CPU and GPU models. We present release dates and benchmark scores of CPU and GPU in Figure 3 and list top models in Appendix B.1. Unless other specified, all measurements in this paper are analyzed from this dataset.

According to our measurements, among all components in the pipeline, the network, queuing (at the decoder queue), and decoding delay are >10ms at the 99th percentile. We highlight them in red in Figure 2. The tail of the application and encoding delay is light since they are processed on commercial servers, which are stable compared to networks and heterogeneous clients. Therefore, we focus on the network, queuing, and decoding delay in the following discussion. We leave the measurement results to Appendix B.2.

We investigate how these three components contribute to the increase of total delay at the tail. For each frame, we denote N, Q, D, and T as the network, queuing, decoding, and total end-to-end delay. We then calculate the conditional probability of $P(X > X_{th} | T > T_{th})$ for each $X \in \{Q, D, C\}$ from our measurements, where X_{th} and T_{th} are thresholds for statistics. A high conditional probability suggests that the component is more likely the cause of $T > T_{th}$. We calculate the conditional probability with different thresholds, and present the results for

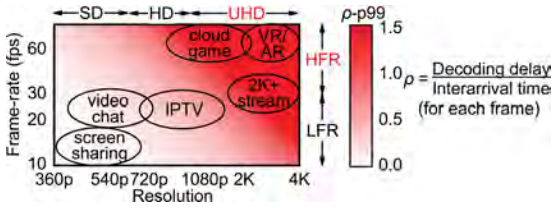


Figure 5: Illustration of the 99th percentile of the utilization ρ of the decoder queue. For high-quality RTC applications (in the top-right corner), the decoder queue is heavily loaded at the tail (shaded red), resulting in an increase of queuing delay at the tail.

network delay and queuing delay in Figure 4.

As we can see, when analyzing the root causes of frames with T>200ms for traditional RTC services, network delay has a high probability (shaded red) to be blamed. However, when analyzing the frames with T>100ms, queuing delay dominates the increase of total delay. Our measurements show that among all frames with an end-to-end total delay of more than 100ms, queuing delay increase happens more frequently than all other component delays: 57% of them have a queuing delay of more than 50ms (stars in Figure 4). Considering the stringent delay requirement of ∼100ms for high-quality RTC, the increase in queuing delay plays a dominant role.

Root cause: The UHD resolution and HFR jointly contribute to the increase in queuing delay. Compared to LFR streaming, HFR increases the arrival rate of the decoder queue by reducing the interarrival time between frames. Also, UHD decreases the departure rate compared to SD streaming by increasing the decoding delay of each frame.

Specifically, we illustrate how the frame rate and resolution could affect the load of the decoder queue by presenting the 99%ile queue utilization in Figure 5. We scale the distribution of interarrival time and decoding delay from our measurements to other frame rates and resolutions. As we can see, for traditional RTC services (the down-left corner), due to their low frame rates and resolutions, the decoder queue still has a utilization of $\rho \ll 1$ at the tail. However, for high-quality RTC applications (the up-right corner), the decoder queue would be heavily loaded, leading to a drastic queuing delay.

The issue is the inconsistency of the decoder's performance *on average* and *at tail*. In fact, many of the hardware decoders that we measured claim to support UHD and HFR videos (e.g., Nvidia GTX series in Table 4). However, according to our measurement, supporting UHD and HFR does not really mean *consistently* supporting. For example, the decoding delay can fluctuate due to numerous reasons including overheating at the client [64], CPU scheduling (§5.1), and the prediction errors [48], all of which are difficult to control for an application. From our measurement with devices in production, the decoding delay is 18ms at the 99th percentile even with hardware acceleration (Appendix B.2). Note that at the frame rate of 60fps, the interarrival time between frames is 16.7ms, resulting in a heavily loaded decoder queue at the tail.

We further analyze the necessity and sufficiency between the

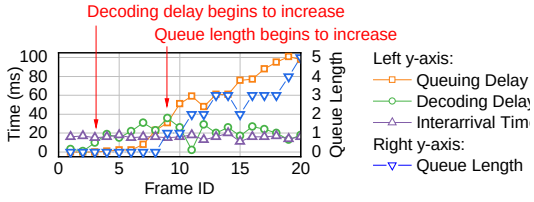
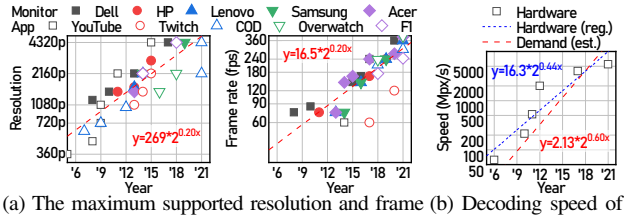


Figure 6: A trace for the accumulation of decoder queue. Note that this is an illustrative example – the distribution of all traces can be found in Appendix B.4.



(a) The maximum supported resolution and frame rate for the top 5 monitor vendors, two streaming platforms (YouTube and Twitch) and three games (Call-of-duty, Overwatch, and F1) [10].

(b) Decoding speed of existing hardware and required decoding speed from demands.

Figure 7: Decoding hardware cannot keep pace with the rapid increase of demands of videos with high resolution and frame rate. Note that the required decoding speed from demands is the frame rate times the *square* of resolution times the aspect ratio.

increase of other components and total delay in Appendix B.3 and figure out that the minor fluctuation of decoding delay leads to the increase of queueing delay. From the queuing theory, when the queue is heavily loaded, the queuing delay will drastically increase [32]. This is because while the decoding delay is continuously fluctuating, the queuing delay is accumulating all the fluctuations of precedent frames. Especially in heavy traffic, a minor fluctuation of the decoding delay could result in a magnitude increase in queuing delay. We refer the readers to [32] for more theoretical analysis. Illustratively, we present a trace from our production service in Figure 6. In the trace, the interarrival time is 16ms, and the decoding delay is 18ms, while the queuing delay is 54ms on average. The continual increase of the decoding delay, although not much by magnitude (18ms) and not long by duration (20 frames, approximately 0.3s), leads to a drastic queuing delay. If such a trace happens with a probability of 1%, we will have a 99th percentile decoding delay of 18ms, and a 99th percentile queuing delay of 55ms. In this case, the tail queuing delay is much higher than the decoding delay, which also contributes to more than half of the end-to-end stutters as analyzed in §3.1.

Trend: hardware decoders cannot keep pace with the increasing demands of UHD and HFR video. User demands for video have increased sharply, as shown in Figure 7(a). For example, the highest supported resolution and frame rate of YouTube have increased from 360p@30fps (7Mpx/s) in 2005 to 8K@60fps in 2015 (2Gpx/s), doubling every 14 months on average. Emerging services at 16K [85, 62] or 240fps [73] further indicate the future demands of UHD and HFR streaming.

However, the decoding speed of the hardware is not increasing as fast. We summarize the decoding speed of state-of-the-art video decoders from recent academic papers [53, 30, 90, 91, 89, 85]. As shown in Figure 7(b), the decoding speed of the state-of-the-art decoding hardware doubles only approximately every 27 months (blue dotted line). Meanwhile, we also calculate the required decoding speed from the existing demands of videos by multiplying the estimated resolution and frame rate from Figure 7(a) and plot the estimation in red in Figure 7(b). The required decoding speed from demands, doubling every 20 months, (red dashed line) increases much faster than the development of decoding hardware (blue dotted line), indicating the continuous incapability of decoding hardware for UHD and HFR videos.

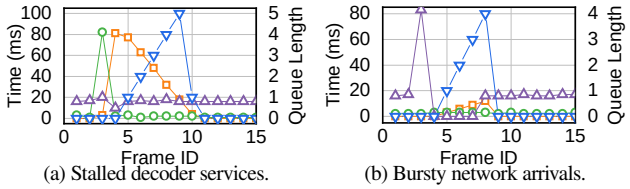
In addition to the state-of-the-art hardware, there are still a considerable number of low-end and mid-end devices in our users. User devices, even in the same generation, could also be very heterogeneous. For example, in Figure 3, notice that the performance of Intel Iris Xe is 2× better than Intel UHD 770 even though the latter is more recent. Thus, there is heterogeneity in user devices even in the same generation. Moreover, new video codecs (e.g., H.265), although with a higher compression ratio, even slow down the decoding speed by up to 60% [24, 55, 21]. In this case, the mismatch between the decoder and UHD and HFR videos will further exacerbate, making the queuing delay at the tail a lasting issue.

3.2 Choice: Controlling Proper Parameters

We motivate the need to adjust the frame rate. For an encoder, there are three parameters that could be independently set, including the frame rate, bit rate, and resolution. The encoder will automatically optimize other parameters (e.g., quantization parameters) based on current contents to achieve the target frame rate, bit rate, and resolution. We refer readers to [17] for more details on video codec.

We first analyze how these parameters could affect the delay of different components. When the bit-rate increases, the network delay will increase due to the congestion. When the resolution increases, since the decoder needs to decode frames with larger pixels, it needs a longer time to decode. The queuing delay depends on the enqueue rate (i.e., frame-rate) and the dequeue rate (i.e., decoding delay). In contrast, for example, if the bit-rate decreases, yet the resolution is kept the same, the decoding delay for each frame will hardly decrease due to the hardware design of the codec, which we further measure in Appendix B.4. Thus, relying on the total delay (e.g., Salsify [34]) would lead to ambiguity in taking effective actions to reduce the delay.

Therefore, we need to individually control respective parameters to reduce different delays. In response, we adjust the frame rate to control the queuing delay for high-quality RTC. When the fluctuations of the decoder and network result in an increase of queuing delay, it is essential to adjust the encoding parameters to reduce the queuing delay. In this case, after collecting measurements from the client and network, the encoder at the server could accordingly adjust the frame rate for the following frames. We could dynamically specify certain timestamps where new frames are encoded.



(a) Stalled decoder services. (b) Bursty network arrivals.

Figure 8: Two traces of transient fluctuations of the decoder queue from online traces. Legends are the same as Figure 6.

We further discuss several potential solutions and concerns of adapting frame rates in Appendix A. In summary, adjusting the resolution or dropping frames is impractical due to the significant overhead of bandwidth. Statically choosing the frame rate based on the client model is also insufficient due to the fluctuation of decoding delay in the runtime. Moreover, since applications have limited control over users' systems, it is also impractical to control the user's system (e.g., pinning the application to a CPU core) for a large-scale production-level service [14]. In terms of frame-rate adaption, note that there are previous efforts in the adaption of frame-rate (e.g., CU-SeeMe [38] decades ago). However, as we discussed in §3.1, with the increase in resolution and frame-rate, and the stringent delay requirements, we need to reemphasize the significance of adapting frame rate now. We also show that it is timely enough to control the frame rate over the Internet.

3.3 Challenges

Achieving an ultra-short queue. To achieve an ultra-short queuing delay for the decoder queue, it is challenging to pick the appropriate indicator to inform the controller when it needs to take action. Existing signals (queue length [60] or queuing delay [77, 34]) fail to achieve an ultra-short queuing delay. Since the accumulation of the decoder queue is the consequence of the fluctuation of the arrival or departure process, both the queue length and queuing delay can only be observed when the queue has already been built up. For the example in Figure 6, while the decoding delay starts to increase at the 3rd frame, a non-zero queue length can only be observed by the 9th frame. We also evaluate baselines based on queue length and queuing delay in §5.2.

In response, we want to capture the *earliest signal* to perceive the potential queuing delay. Therefore, instead of measuring the queuing delay, we want to estimate the potential increase of queuing delay predictively. For example, inspired by recent advances in congestion control [36, 50], a straightforward way is to measure the dequeue rate of the decoder queue to estimate the potential increase of the queuing delay.

However, in terms of tails, the arrival process is also fluctuating, which could also lead to an increase in queuing delays. For example, the network delay might increase by ten times at the 99th percentile than the median [25]. In response, to precisely avoid queue accumulation, we extend the designs of [36, 50]: AFR comprehensively measures the arrival and departure process and controls the queuing delay based on queuing theory. We introduce the design in §4.2, and evaluate the necessity of measuring the arrival process in §5.2.

Handling various events. Furthermore, the reason behind the formulation of the decoder queue in high-quality RTC is complex. As we introduced in §3.1, the stationary degradation of decoding capacity could lead to the accumulation of the decoder queue, e.g., the traces in Figure 6. Besides, the decoder queue could also be accumulated due to transient contingencies. For example, from our experiences in production, the decoder might contingently experience a sudden decoding lag of $\sim$100 milliseconds (e.g., the 3rd frame in Figure 8(a)). The sudden interference in wireless channels might also lead to the bursty arrival of several frames (e.g., the 4th to 8th frames in Figure 8(b)). In both cases, the decoder queue will be accumulated. Since these transient fluctuations happen suddenly, it is challenging for the controller to react by measuring enqueue and dequeue rates.

Algorithm 1: Hierarchical AFR control.

Input: Enqueue process $\{A_n\}$, dequeue process $\{S_n\}$, queue states Q. (A_n denotes the interarrival times, and S_n denotes the decoding delays of frames $\{n\}$.)
Output: Target frame rate f.
1 $f_0 =$ StationaryController($\{A_n\}$,$\{S_n\}$)
2 $\alpha =$ TransientController(Q)
3 $f = \alpha f_1$

Thus, AFR differentiates the causes of queue accumulation and reacts respectively to fluctuations at different time scales. We design a stationary controller to avoid queue accumulation in heavy traffic (§4.2), and a transient controller to reduce the queuing delay in contingencies (§4.3).

4 Design – Adaptive Frame-Rate (AFR)

We first analyze the overall workflow of AFR in §4.1, and then present the two controllers of AFR (§4.2, §4.3).

4.1 Workflow Overview

The workflow of AFR is presented in Algorithm 1. Specifically, the stationary controller (§4.2) maintains the queue around an ultra-short target based on dynamics of enqueue and dequeue processes. By measuring the statistics of both processes, AFR calculates the expectation of the queuing delay based on queuing theory. The frame rate can therefore be optimized towards a given queuing delay target (line 1). The transient controller observes the queue states Q (queue length and queuing delay) and calculates the discounting factor $\alpha \leqslant 1$ (line 2) to further decrease the frame rate when the queue formulates. The final frame-rate is the stationary frame-rate f_0 discounted by α (line 3). In this case, AFR can react to various scenarios of queue accumulation.

4.2 Stationary Controller

As introduced above, we measure the arrival and service processes and control the expected queuing delay of the queue. Specifically, we use the Kingman formula as an approximation of the expectation of queuing delay. Kingman formula is a widely adopted approximation formula of queuing delay [45] for G/G/1 queues. Compared to other approximation methods, in this paper, we adopt the Kingman formula to estimate the queuing delay since its estimation is from *both arrival and departure processes*

without relying on queue states, which could provide the earliest signal for the potential queuing delay. According to the Kingman formula, the expectation of queuing delay τ_{queue} follows:

$$\mathbb{E}(\tau_{queue}) \approx \left(\frac{\rho}{1-\rho}\right)\left(\frac{c_a^2+c_s^2}{2}\right)\mu_s \tag{1}$$

where

$$c_a = \sigma_a/\mu_a, \quad c_s = \sigma_s/\mu_s, \quad \rho = \mu_a/\mu_s \tag{2}$$

(μ_a, σ_a) and (μ_s, σ_s) are the mean and standard deviation of the arrival and service processes:

$$\mu_a = \mathbb{E}\{A_n\}, \sigma_a = \sqrt{var(A_n)}, \mu_s = \mathbb{E}\{S_n\}, \sigma_s = \sqrt{var(S_n)} \tag{3}$$

From Eq. 1, the queuing delay is related to the following factors:

- Queue utilization ρ. The queuing delay will increase when the queue is overloaded ($\rho \to 1$). The current frame rate and decoding delay determine the queue utilization.
- Arrival and service fluctuations c_a and c_s. When the arrival or the service processes fluctuate, the queuing delay will also increase.
- Service time μ_s. Finally, the queuing delay scales with the average decoding delay.

Therefore, we control the expected queuing delay by controlling the right-hand side (RHS) of Eq. 1. We set $\mathbb{E}\{\tau_{queue}\}$ to a pre-defined queuing delay target W_0. Consequently, the target frame-rate f_0 could be calculated as:

$$f_0 = \rho/\mu_s = 1/\left(\mu_s \cdot \left(1 + \frac{\mu_s}{W_0} \cdot \frac{c_a^2+c_s^2}{2}\right)\right) \tag{4}$$

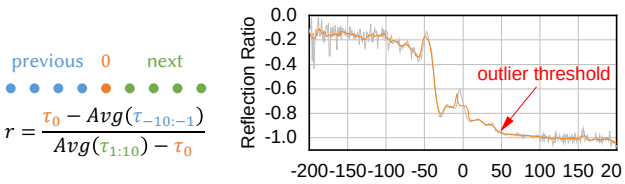
Discussion: Approximation method. The AFR mechanism supports any approximation formula by design. There are other research efforts to control the queue. For example, recent efforts in congestion control [36, 50] directly set the target utilization (e.g., setting $\rho = 0.95$) and calculate the enqueue rate. In this paper, we adopt Kingman formula to capture *both the arrival and departure processes*, as discussed in §3.3. We also evaluate the performance of other baselines in §6.1.

Measurements of queuing dynamics. According to Eq. 4, we need to measure the mean and variance of the arrival and service processes. Similar to the RTT measurements in TCP [44], we adopt the exponentially weighted moving average (EWMA) and exponentially weighted moving variance (EWMV) to estimate the $\mu_s, \sigma_s, \mu_a, \sigma_a$ in Eq. 1 and 2.

$$\begin{aligned} \hat{\mu}_n &= \xi_\mu x_n + (1-\xi_\mu)\hat{\mu}_{n-1} \\ \hat{\sigma}_n &= \sqrt{\xi_\sigma (x_n - \hat{\mu}_n)^2 + (1-\xi_\sigma)\hat{\sigma}_{n-1}^2} \end{aligned} \tag{5}$$

where x_n denotes interarrival time A_n or service time S_n. $\hat{\mu}_n$ and $\hat{\sigma}_n$ are the EWMA and EWMV. ξ_μ and ξ_σ are the discounting factors for the measurement of mean and standard deviation, trading off between precision and sensitivity.

However, due to bursty arrival or stalled services (§4.1), both the arrival and service processes could have significantly deviated value. For example, the 3rd frame in Figure 8(a) has a decoding time of 82ms while other frames are below



(a) Illustration (b) Measurements in production

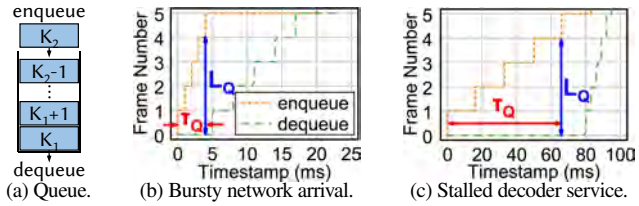
Figure 9: Reflection in outlier removal. Figure 9(b) presents the frequency of frames with $r \in [\frac{1}{C}, C]$. Measurement details in §5.2.

4ms. Such outliers will significantly deviate the estimation of stationary statistics for a long period. In fact, as we discussed in §4.1, these contingent events are designed to be handled by the transient controller. Therefore, we need to filter those outliers out to precisely estimate the stationary status of arrival and service processes. Due to the highly skewed distribution of decoding delay, existing outlier removal mechanisms based on standard deviation (e.g., the three-σ rule [65, 68]) suffer from differentiating stationary state transitions from outliers.

To capture the transitions of the status of decoders while eliminating the influence of the contingent outliers, we introduce an outlier removal mechanism based on priori knowledge from measurements in production. The key intuition is that *decoding delay differences* ($S_n - S_{n-1}$) are related to the probability of being outliers. For example, an increase of 20ms on decoding delay is probably the transition between stationary states (Figure 6). However, a sudden increase of 80ms on decoding delay is likely to indicate that decoding delay is an outlier, which is usually the scenario of contingent stalls in Figure 8(a). This is because commercial decoders are usually able to decode frames at the frame rate of 24fps on average. According to our measurements, when the decoding delay difference is above 50ms, the possibility of being an outlier for that frame is 95%. Thus, we remove frames with a decoding delay difference of >50ms in the stationary controller, and leave the control of those frames to the transient controller.

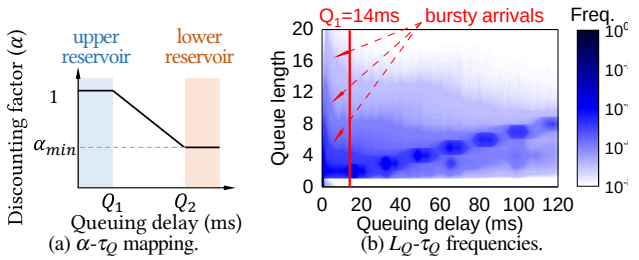
We further characterize our observation based on measurements in production. As shown in Figure 9(a), we quantify the outlier with *reflection ratio r*, which illustrates the recovery of decoding delay before and after the potential outlier. The numerator is the difference between the current decoding delay (τ_0) and the average decoding delay of the previous 10 frames ($\tau_{-10:-1}$), and the denominator is the difference between τ_0 and future decoding delay. For outliers of contingent stalled service (e.g., the 3rd frame in Figure 8(a)), their reflection ratios would approach -1. This is because previous frames and subsequent frames have similar decoding delays, while the outlier has a much higher decoding delay ($\tau_0 \gg \tau_{-10:-1} \approx \tau_{1:10}$).

We then plot the relationship between the difference of decoding delay ($\tau_0 - \tau_{-1}$) and the average reflection ratio (r) of all frames with the same difference from our measurements in Figure 9(b). When the decoding time difference is larger than 50ms (marked with a red arrow), the average reflection ratio is less than -0.95, indicating that most frames in this scenario are outliers. Therefore, the stationary controller in AFR does not calculate the



(a) Queue. (b) Bursty network arrival. (c) Stalled decoder service.

Figure 10: Differences between bursty network arrivals and stalled decoder services. The y-axis is the accumulated enqueue/dequeue frames. For example, the enqueue curve in Figure 10(b) increases from 1 to 2 at 1ms, indicating that frame #2 enqueues at 1ms.



(a) α-τ_Q mapping. (b) L_Q-τ_Q frequencies.

Figure 11: Illustrations and measurements of the transient controller. A series of linearly distributed dark blue clusters in Figure 11(b) indicate that L_Q and τ_Q are linearly correlated.

frames with a decoding delay difference larger than 50ms.

Convergence time analysis. To help operators to better understand the behavior of the stationary controller, we investigate the convergence of the stationary controller during state transitions of the service process. We want to answer the following question: During the transition from stationary state (μ_1,σ_1) to (μ_2,σ_2), how long will the stationary controller take to converge to the new frame-rate and drain the potential accumulation of the queue due to the transition?

We outline the main conclusion here and leave the detailed analysis in Appendix E. When the control loop (round-trip delay) of AFR is τ frames, the convergence time T_0 is bounded w.r.t. τ and W_0, and is acceptable for most scenarios. For example, when the average control loop of AFR is the interarrival time of one frame (τ=1), and W_0=2ms, the stationary controller could converge to the new stationary state within 2 frames. We illustrate the convergence time of the stationary controllers with more settings in Appendix E.

4.3 Transient Controller

The transient controller is designed to handle the contingent queue accumulations (§4.1). Therefore, we need to first understand how we should react to these queue contingencies.

Understanding queue contingencies. As shown in Figure 8(a) and 8(b), both stalled decoder services and bursty network arrivals will cause a sudden increase in queue length. We illustrate the enqueue and dequeue events of two contingencies in Figure 10. In Figure 10(b), 5 frames arrive at the client together within 4ms, resulting in a queue length of 4 when the 5th frame arrives and observes, as illustrated with the L_Q (blue arrow). In Figure 10(c), the decoder takes 80ms to decode the 0th frame, when queued frames cannot be dequeued to the decoder. Therefore, upon the arrival of the 5th frame, it also observes a queue length of 4.

However, the bursty network arrivals and stalled decoder services should be handled separately. In the scenario of bursty network arrivals, the bottleneck of total delay is still in the network due to its long network delay. As long as the decoder is functional, even if multiple frames arrive at the queue simultaneously, they could be processed efficiently (Figure 8(b)). In this case, the queue will be drained in a short time, and we do not need to reduce the frame rate. In contrast, the stalled decoder service will drastically increase the queuing delay of subsequent frames and needs adaption (Figure 8(a)). Thus, we need to differentiate between the two scenarios.

Since both scenarios result in an increase in queue length, they cannot be effectively differentiated with queue length only. Our insight is that we can differentiate them with the sojourn time of the first frame in the queue. As shown in Figure 10(a), at the arrival of frame K_2, the sojourn time τ_Q of the first frame K_1 and queue length L_Q observed by K_2 are:

$$\tau_Q = t_{enq}^{(K_2)} - t_{enq}^{(K_1)}, \qquad L_Q = K_2 - K_1 \tag{6}$$

where $t_{enq}^{(i)}$ is the enqueue timestamp of frame #i, and frame #K_1 is the frame at the head of the queue. For bursty network arrivals, since frames arrive at the decoder queue simultaneously, when the last frame of the burst arrives, the first frame has only been queued for a short time. For example, τ_Q in Figure 10(b) is 4ms (marked red). In contrast, for stalled decoder service, the head frame has been blocked for a long time, leading to a high τ_Q of 66ms in Figure 10(c). Therefore, we use τ_Q to adjust the frame rate in the transient controller.

Feedback control. For the transient controller, the design space is to find out a mapping between the discounting factor α and the queuing delay τ_Q. Since the transient controller is designed to reduce the frame rate based on the results of the stationary controller, the possible range of α satisfies:

$$f_{min}/f_{max} = \alpha_{min} \leqslant \alpha \leqslant 1 \tag{7}$$

where f_{min} and f_{max} are the lower and upper bounds for frame rate required by the application. Since longer τ_Q indicates a more severe load of the queue, the discounting factor should decrease with the increase of τ_Q. Besides, the α-τ_Q mapping should also have the following properties:

First, avoid overreactions. As we discussed above, for bursty network arrivals, τ_Q will also slightly increase due to the volumetric arrived frames. However, since such a transient queue accumulation will be cleared quickly as long as the decoder is functional (Figure 10(b)), we should not decrease the frame rate. Therefore, we need to introduce an upper reservoir (as shown in Figure 11(a)) to avoid overreactions. In the upper reservoir, when a non-zero but small τ_Q is observed ($0 \leqslant \tau_Q \leqslant Q_1$), the transient controller will not decrease the frame rate. The reservoir threshold Q_1 should be set based on measurements. We measure the observed L_Q and τ_Q from frames and present the results in Figure 11(b). Peaks near the left axis (marked by red dashed

arrows) represent frames with a long L_Q yet with a short τ_Q, which are due to the bursty network arrivals. Therefore, we set Q_1 to filter out those bursty arrival-related peaks (e.g., Q_1=14ms in our deployment, the red line in Figure 11(b)).

Second, respond timely. Due to the stringent delay requirements of high-quality RTC applications, a long queuing delay will drastically degrade the users' experiences. Therefore, we need to control the slope of the mapping in Figure 11(a) to effectively reduce the queuing delay. Since α is lower bounded, we could control the slope of the mapping by introducing a *lower reservoir*, as shown in Figure 11(a). We set Q_2 as the maximum tolerable queuing delay:

$$Q_2 = \max(Q_1, Deadline - \tau_{network} - \tau_{decode}) \quad (8)$$

where $\tau_{network}$ is the round-trip network delay, and τ_{decode} is the decoding delay μ_s. *Deadline* is the requirement for the total delay of the application. Based on users' experiences in the human-machine interaction and our operational experiences, we set *Deadline* to 100ms in our deployments [43].

5 Implementation

We implement the AFR with a frame-level trace-driven simulator, and deploy the AFR onto a production high-quality RTC service in the wild. In this section, we present the design of our simulator (§5.1), introduce the simulation setup (§5.2) and the deployment setup (§5.3).

5.1 Simulator Design

To faithfully compare and replay the traces for different queue control algorithms, we design a simple simulation environment that models the dynamics of RTC. The simulator maintains the decoder queue and replays the traces collected from online services, where the traces contain the decoding delay, network delay, original queuing delay, and also the arrival timestamp for each frame. Specifically, frames arrive at the decoder queue according to timestamps in traces, wait in the decoder queue for dequeuing, and are decoded according to decoding delays in traces. To avoid frequently sending frame-rate adjustment requests to the servers, frame rates are quantized at the level of 5fps, which is also followed by our online deployment. We implement the potential interference from CPU time-slicing: since the fetching of frames to decoders depends on the CPU, there are possible cases where fetching the frame from the queue to the decoder needs waiting to be scheduled by the CPU by up to several milliseconds [20]. Therefore, we further profile such a delay in the traces and introduce the scheduling waiting time in our simulator. We also implement the response time of the encoder between the new frame-rate actions and new frames generated with the updated frame rate, according to our measurements in §6.4. Please refer to Appendix C for implementation details.

5.2 Simulation Setup

Traces. We measure the frame-level statistics of our cloud gaming service (introduced in §3.1) on two types of clients (Windows and MacOS) and access networks (Ethernet and WiFi). We profile each step of received frames in one of our production clusters for 24 days in December 2020. This results in a dataset with 7.73 billion frames and 41.7k hours of playtime (Table 1), which is the largest frame-level dataset for interactive streaming to the best of our knowledge.

	Category	Session	Frame	Playtime
(1)	Windows+Ethernet	29.7k	6.35 B	34.2k hours
(2)	Windows+WiFi	6.4k	1.12 B	6.2k hours
(3)	MacOS+Ethernet	0.4k	40.9 M	0.2k hours
(4)	MacOS+WiFi	2.1k	216 M	1.1k hours
	Total	38.1k	7.73 B	41.7k hours

Table 1: Distribution of our traces on the client type.

Parameter settings. There are several parameters in AFR to be determined. Except for the parameters related to the transient controller (§4.3), we set W_0 in the stationary controller to 2ms and the discounting factors in EWMA $\xi_{arrv} = 0.033$ and $\xi_{serv} = 0.25$. We discuss the sensitivity of those settings and their influence on the performance in §6.3.

Metrics. In the evaluation, we mostly measure the delays (including the queueing delay and the end-to-end total delay). As we discussed in §2, the delay in interactive streaming is orthogonal to other video quality metrics (e.g., PSNR [2] or SSIM [76]). The delay, which represents interactivity, is the main optimization goal in this paper. We demonstrate that AFR has negligible degradation on the video quality in §6.4.

Baselines. To evaluate the performance of AFR, we implement existing frame control mechanisms as follows:

- `DropTail` is the frame control mechanism in WebRTC [60]. When frames overflow the queue, the client will first clear the queue, then request a new key frame, and finally drop all frames until the next key frame arrives. We set the queue capacity to 16 frames.
- `QLen-S` observes the current queue length, skips frames from the content generator before the encoder if the queue length is $\geqslant 1$, and resumes if the queue length is <1.
- `QWait-S`. We migrate the frame control mechanisms from existing academic efforts in our simulator [34, 77], and replace the signal from total delay to queuing delay to better reduce the queuing delay. Since these baselines are not designed for stringent delay requirements of 100ms, we also finetune their parameters with our traces. `QWait-S` skips frames before the encoder if the queuing delay is $\geqslant$32ms, and resumes if the queuing delay is $<$4ms.

Besides, to evaluate the effectiveness of different components in AFR, we also different variants of AFR:

- `AFR-QLen`. We demonstrate the insufficiency of controlling the frame rate with queue states with a feedback algorithm based on current queue length: it observes the current queue length at the arrival of each frame, and maps the queue length of$\{0, 1+\}$ to frame-rate$\{60, 24\}$fps.
- `AFR-QWait`. A feedback algorithm maps current queuing delay of$\{(0, 4), (4, 8), (8, 12), (12, \infty)\}$ms to frame-rate of $\{60, 48, 36, 24\}$fps. The parameters have also been finetuned with our traces.

- `AFR-TX`. To demonstrate the effectiveness of measuring both the arrival and service process, we further implement a dequeue rate-based algorithm. `AFR-TX` measures the dequeue rate and sets the target frame-rate with $\rho = 0.8$, where ρ has been tuned with our traces. The dequeue rate is the reciprocal of decoding delay.
- `AFR-Kingman`. Moreover, we individually evaluate the stationary controller of AFR to further illustrate the effectiveness of the transient controller.
- `AFR`. Finally, we put all optimizations in this paper (both the stationary and transient controller) together.

We present how we tune the parameters, and evaluate the trade-offs between frame rate and queuing delay in §6.3.

5.3 Deployment setup

We finally deploy AFR onto our cloud gaming service. The gaming service X employs the H.264 codec to increase the coverage of hardware decoding and adaption towards heterogeneous clients[1], and customizes the codec performance for the optimization of gaming. Tencent Start currently supports 13 production-level games, including action-adventure, first-person shooter, and real-time strategy games. To optimize the network delay, the service is accelerated with multi-access edge computing similar to [74, 57, 86]: Users are split into tens of operation regions with a geographical diameter of hundreds of kilometers. Cloud gaming servers are deployed on clusters in each operation region, resulting in an average round-trip network delay of 15ms (Appendix B.2).

The frame-rate adaption algorithms are implemented on the client side. The AFR controller continuously measures the statistics of the decoder queue, and sends requests to edge servers to adjust the frame rate when necessary. The edge server then forwards the frame-rate adjustment requests to both the video encoder and the gaming application. New frames will be generated following the new inter-frame interval. We evaluate the response timeliness and overhead of video encoder and gaming application in §6.4.

6 Evaluation

We evaluate the AFR controller in the following aspects:

- **Delay improvements.** We present the performance improvements: The ratio of frames with long queuing delay and total delay of AFR has been improved by 2.1×-26× and 13%-2.2× against existing baselines (§6.1).
- **Frame-rate maintenance.** We then demonstrate that AFR introduces negligible impacts on the metrics related to frame-rate (§6.2).
- **Parameter sensitivity.** Our evaluation shows that parameters in AFR have a wide range of settings to gain performance improvements against finetuned baselines (§6.3).
- **Microbenchmarking.** We further demonstrate that the timeliness, overhead, and image quality of frame-rate adjustments are satisfactory for online deployment (§6.4).

[1]Hardware decoding has a shorter decoding delay than software decoding and supports higher frame rates. H.264 has a higher coverage of hardware decoding support compared to other advanced codecs [54].

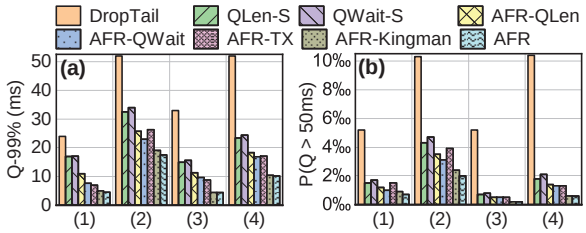


Figure 12: Simulation results of queuing delay (the 99%ile and the ratio of frames with >50ms queuing delay).

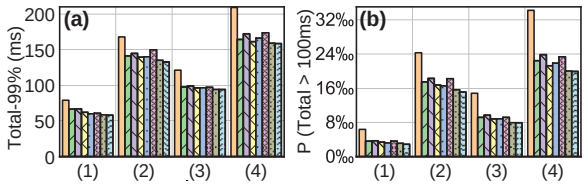
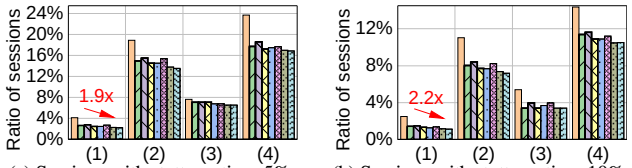


Figure 13: Simulation results of total delay (the 99%ile and the ratio of frames with >100ms total delay).



(a) Sessions with stutter ratio >5%. (b) Sessions with stutter ratio >10%.

Figure 14: Ratio of sessions with different stuttered frames.

- **Deployment in the wild.** Finally, we report the A/B test results and the deployment progress of AFR on our cloud gaming service online (§6.5).

6.1 Delay Improvements

We compare the queuing delay and the total delay of each frame with AFR and baseline algorithms in four sets of traces (Table 1). We measure the queuing delay in two dimensions: we present the 99th percentile queuing delay and the ratio of frames with a queuing delay >50ms in Figure 12. We first analyze the results of AFR against three existing mechanisms (`DropTail`, `QLen-S`, and `QWait-S`). AFR could reduce the 99%ile queuing delay by 1.9× to 7.4×, and the ratio of severely queued frames by 2.1× to 26× on different sets of traces against three baselines. In this case, the 99%ile queuing delay could be squeezed to 6.9ms. This indicates that AFR could effectively achieve an ultra-short queuing delay. AFR also demonstrates satisfactory performance improvements on the *total end-to-end delay*, which is directly related to users' experiences. AFR improves the 99%ile total delay by 27% to 36%, and the ratio of severely delayed frames (total delay >100ms) by 1.6× to 2.2× in all traces. We also measure the *session stutter ratio*, i.e. the ratio of frames with a total delay of >100ms in a session, for each session. We then measure the ratio of sessions with a session stutter ratio of >5% and >10%, which indicates how many users suffer from unsatisfactory experiences and present the results in Figure 14. For the major population of our service (Cat. (1), Table 1), AFR reduces the stuttered sessions by 17% and 21% compared to the best of the three baselines. For other categories, the ratio of

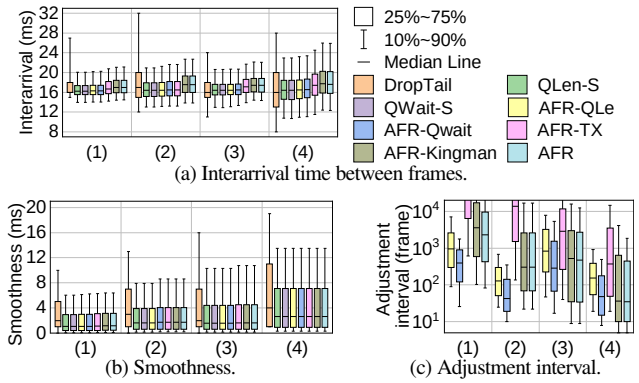


Figure 15: Frame-rate maintenance. Better viewed in color.

stutter sessions has also been reduced by 5% to 37%. AFR could significantly improve experiences for high-quality RTC.

We further understand the performance improvements with the comparisons among different variants of AFR. Compared to `DropTail`, baselines based on queue states (`AFR-QLen`, `AFR-QWait`) could effectively reduce the queuing delay, indicating the necessity of actively controlling the queuing delay (§3.1). Compared to `QLen-S` and `QWait-S`, controlling the frame rate achieves better performance than skipping frames from the encoder. This is because skipping frames would drastically degrade the tail frame rate, for which the parameters of baselines are tuned (§6.3). `AFR-TX` could further reduce the queuing delay than the queue state-based baselines, indicating that observing the service process could know the potential degradation in advance and effectively take actions, validating our analysis in §3.3. `AFR-Kingman` further improves the performance by 10% against `AFR-TX`, demonstrating that the fluctuating arrival of the high-quality RTC could also affect the estimation of the decoder queue. `AFR` finally reduces the tail queuing delay by 2-4% against `AFR-Kingman`, indicating the necessity of the transient controller to handle contingencies.

Besides, we also find that AFR has larger performance improvements when the network is better. The performance improvements on two sets of Ethernet traces (55% and 37% for Cat. (1) and (3)) are larger than the on WiFi traces (35% and 27% for Cat. (2) and Cat. (4)). Considering the ongoing deployment of next-generation access networks with better network conditions (e.g., 5G and WiFi 6), the necessity of controlling the decoder queue would be more significant.

6.2 Frame-rate Maintenance

Besides, we also measure the effect of AFR on the frame rate. We first measure the interarrival time between frames at the arrival of each frame on the client. For example, a frame rate of 60fps should result in an interarrival time of around 16.7ms. We tune the parameters of each algorithm to keep the 99th percentile of their interarrival time at the same level (details in §6.3). Therefore, for 10-90th percentiles, as shown in Figure 15(a), most algorithms except for `DropTail` are comparable. Compared to the existing deployed mechanism `DropTail`, AFR even improves the tail user-perceived frame rate due to its better management of frame

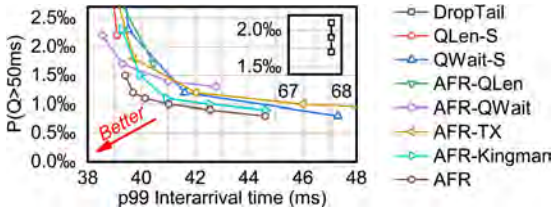


Figure 16: The trade-off between the tail interarrival time and queuing delay. We tune the parameters for baselines and AFR to illustrate the capability of each algorithm in the trade-off.

drops. AFR slightly decreases the median frame rate by 3%-9%, which brings the negligible quality of experience (QoE) degradation to users considering the improvements on delay [71, 81].

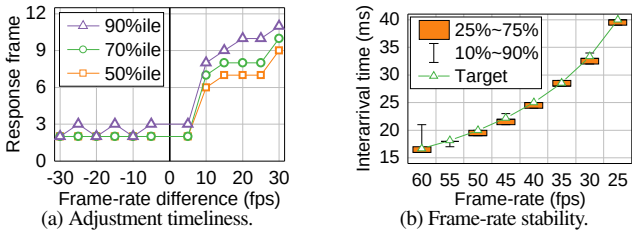
We further measure the smoothness of frame-rate, which might also have potential effects on users' experiences [33]. We measure the *differences of interarrival time* as an indicator of the smoothness of frame rate and present the results in Figure 15(b). Except for `DropTail`, all baselines and AFR have similar interarrival differences and are better than `DropTail`. This is mainly because that frame drops in `DropTail` will introduce a sudden increase of interarrival differences. Moreover, we also measure the frame adjustment interval and present the distributions in Figure 15(c). The median adjustment interval of AFR is hundreds to thousands of frames, which is much longer than the response time of frame-rate adjustment (§6.4).

6.3 Parameter Sensitivity

We then evaluate the sensitivity of parameters in AFR and other baselines. We tune parameters of all baselines in §5.2: thresholds for skipping frames for `QLen-S` and `QWait-S`, mappings for `AFR-QLen` and `AFR-QWait`, ρ for `AFR-TX`, and W_0 for `AFR-Kingman` and AFR. We present the ratio of frames with queuing delay >50ms (P(Q>50ms)) and the 99th percentile of interarrival time on Cat. (1) traces in Figure 16. The down-left corner indicates the algorithm has a satisfactory trade-off between the queuing delay and the frame rate.

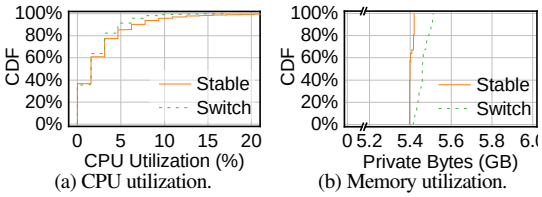
As we can see, AFR outperforms all other baselines in a wide range of settings, achieving a better trade-off between the queuing delay and frame rate. `QLen`-based algorithms are challenged in achieving ultra-short queuing delay: with the extremest parameters (skipping/decreasing frame-rate as long as queue length is non-zero), `QLen-S` and `AFR-QLen` could only achieve a P(Q>50ms) of 2.2‰ and 1.7‰, much higher than other baselines. This follows our analysis in §3.3 that queue length is too coarse-grained as a signal to control the queue with an ultra-short target. Meanwhile, skip-based algorithms could achieve lower queuing delay compared to frame-rate-based algorithms, yet with higher interarrival time. The parameters of all algorithms are tuned according to Figure 16 by aligning the 99th percentile interarrival time.

We also evaluate how different percentiles of queuing delay and total delay are affected by the setting of W_0 in Appendix D.3. The performance of AFR reacts sensitively to the setting of W_0, indicating that operators could effectively balance the total delay and frame rate by adjusting W_0. We further evaluate



(a) Adjustment timeliness. (b) Frame-rate stability.

Figure 17: Effectiveness of frame-rate adjustment.



(a) CPU utilization. (b) Memory utilization.

Figure 18: Frame-rate adjustment overhead.

the sensitivity of the discounting factors ξ of the EWMA and EWMV in the transient controller (§4.3) in Appendix D.3, demonstrating how operators should set these parameters to balance between the precision and sensitivity.

6.4 Microbenchmarking

We also benchmark AFR in a testbed of our cloud gaming service.

Effectiveness of frame-rate adjustment. We first measure the responsiveness and precision of frame-rate adjustment at the video encoder. We enumerate all frame-rate switching within{25, 30, ···, 60}fps, and measure how many frames the encoder needs to take to steadily output video streams at the new frame rate. The response time measured by the unit of frame (i.e. *response frame*) is presented in Figure 17(a). For each group of settings, we repeat the experiments 100 times to eliminate the randomness. When decreasing the frame rate, the 90%ile response frames is less than 3 frames, indicating the encoder and gaming application could decrease the frame-rate timely. This could effectively alleviate the overload of the decoder queue. When significantly increasing the frame rate, the frame rate might be slightly delayed to change. This is because the frame rate at the client side follows the bucket effect. Either encoder or the gaming application decreases the frame rate will lead to a decrease of the final frame rate, while the increase of frame rate needs an increase from both components. Even so, the tail response frame is $<$10 frames, which is much less than the adjustment interval (Figure 15(c)).

We then measure the fluctuation of the frame rate of the output of the streaming encoder. We set the frame rate to several levels as above, and measure the interarrival time between each frame. For each frame rate, we measure the interarrival time for 30,000 frames and present the distribution in Figure 17(b). The interarrival time between frames largely falls around the target frame rate. Therefore, unlike the fluctuating bit-rates in video streaming [42], frame-rate could be precisely controlled by the encoder.

Frame-rate adjustment overhead. We further measure the potential processing overhead of frame-rate adjustment at the edge server. To magnify the overhead, we change the frame rate

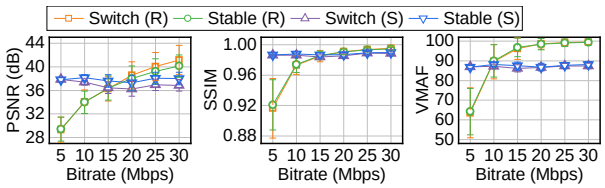


Figure 19: The image quality differences of AFR and the original video tested in a running scene (R) and stable scene (S). The error bar represents the standard deviation.

from 60fps to 30fps and back to 60fps every 6 frames, which is much shorter than the usual adjustment interval. We then measure the CPU and memory utilization of the cloud gaming application and encoder by sampling the CPU processing time and application private bytes with the `typeperf` [66] every 1 second. We measure for 30 minutes to eliminate the randomness. We compare the scenario with a stable frame-rate of 60fps (`stable`) and a frequently switching frame-rate (`switch`) in Figure 18. For CPU utilization, both scenarios have a similar distribution from 0% to 20%. `switch` is a little better than `stable` since producing a lower frame rate takes fewer CPU resources for the gaming application. As for memory utilization, the major memory consumption is from the gaming application. Frame-rate switching slightly increases the utilization of private bytes since frequently resetting the encoder requires allocation of memory. Nonetheless, the increase of memory utilization is less than 1.8% even at the 99%ile, which is negligible and could be even lower in the case of normal frame-rate adjustments.

Image quality degradation. We also investigate the potential image quality degradation caused by AFR. We record two raw videos from games, one in a running scene (R) and another in a standing scene (S). For each video, we switch the frame rate every 100 frames 15 times and measure the video quality for the following 400 frames. We investigate three video quality metrics, peak-signal-to-noise-ratio (PSNR) [2], structural similarity index (SSIM) [76], and video multimethod assessment fusion (VMAF) [51], and present the results in Figure 19. `stable` and `switch` denote the scenarios where the frame-rate remains unchanged or frequently switched. Results demonstrate that frequently switching the frame rate will not affect the video quality: the video quality of two videos on three metrics are comparable in all cases.

6.5 Deployment in the Wild

Finally, we evaluate the performance of AFR by deploying it onto Windows clients of our cloud gaming service, Tencent Start, in one of its production clusters. Before the deployment of AFR, our cloud gaming service follows the frame control strategy in WebRTC (i.e., `DropTail`). To make a clean and controlled comparison, we only present the results from online A/B tests in our production clusters, when all other implementations and settings are kept the same. The A/B test is conducted from January 8, 2021, to January 14, 2021, resulting in 5369 Ethernet sessions and 1467 WiFi sessions. The parameter settings of AFR remain the same as the simulation (§5.2). We randomly enable

Cat. (1)	Q99	Q>50ms	T99	T>100ms	Session
DropTail	54ms	1.11%	101ms	1.03%	7.30%
AFR	22ms	0.51%	80ms	0.68%	5.82%
Cat. (2)	Q99	Q>50ms	T99	T>100ms	Session
DropTail	64ms	1.83%	174ms	3.00%	24.00%
AFR	37ms	0.54%	160ms	2.11%	21.17%

Table 2: Performance of deployment in the wild. Metrics are the 99%ile of queuing delay (Q99), the ratio of frames with Q>50ms, the 99%ile of total delay (T99), and the ratio of the stuttered frame (T>100ms). `Session` is the ratio of sessions with stutter ratio >5%. Cat. (1) and (2) are Ethernet and WiFi on Windows clients.

(or disable) AFR with a probability of 50% for each session, and present the results in Table 2. Similar to the simulation results, the ratio of stuttered frames measured by total delay (P(T>100ms)) in both categories has been improved by 34% and 30%, which significantly improves users' experiences in interactive streaming. The stuttered sessions (with the same metric as Figure 14(a)) have also been reduced by 17% on average, indicating these users could be alleviated from stuttering streaming experiences. Therefore, the online deployment also demonstrates significant benefits of AFR for high-quality RTC users. AFR has already been deployed onto all production clusters of Tencent Start for over one year, serving thousands of users each day.

7 Discussions

In this section, we discuss the potential limitations of AFR.

Application scenarios. In this paper, we mainly evaluate the performance of AFR on traces or production clusters of our cloud gaming service. However, as we introduce in §1 and §2, the overload of decoder queue generally exists in many high-quality RTC scenarios, such as VR streaming or 4K live streaming, as long as they stream high frame-rate and high bit-rate video onto commercial clients. We evaluate AFR with cloud gaming due to access to the real-world traces and production services X. We leave the deployment of AFR over other scenarios as our future work.

Coexistence of multiple control loops. There are other control loops that work simultaneously in the RTC system. For example, the underlying congestion controller will also control the bit-rate of the video based on network conditions [25]. The video codec will also adjust the quantization parameter based on the scenes to encode [17]. As we discussed in §3.2, these parameters are affected by different causes (network congestion, decoder degradation, scene variation), which are orthogonal to each other. Therefore, the adaption of the frame rate is orthogonal to the other controllers in the RTC system. In §6.5, we evaluate the performance of AFR with all these controllers in our real production in the wild. We leave the coordination of different controllers on the joint optimization over the user's experience for the future.

8 Related Work

There has been little prior work on the decoder queue for high-quality RTC. We survey the following three aspects.

Frame controls in RTC. As we discussed, besides the `DropTail` mechanism in WebRTC, there are a series of research efforts in the active control of RTC frames. For example, some work [38, 77, 34] maintains a certain number of in-flight frames based on total delay [77] or frame-level acknowledgement mechanisms [34]. AFR differentiates from them in two aspects. First, the end-to-end control introduces ambiguity in taking effective actions, as discussed in §3.2. AFR takes effective actions to reduce the queuing delay. Second, existing control strategies are based on queue lengths or queuing delay. In contrast, measuring the arrival and service processes in AFR could help high-quality RTC to achieve lower queuing delays. Furthermore, researchers also proposed to co-design the codec and network [35, 34, 79, 75] or even redesign new decoding ASICs [90]. These cannot be accelerated with commercial hardware and are hard to deploy in practice.

Adaptive Bit-rate Control. There have already been a series of research efforts on the optimization of low-latency streaming. Different congestion control [78, 83, 25] or rate adaption algorithms [87, 84] have been proposed to enable the low-latency transport for real-time communications. However, as we discussed above, changing the bit rate without changing the frame rate will not alleviate the load of the decoder queue. Since bit-rate and frame-rate are independent in video streaming, bit-rate adaption is orthogonal to frame-rate adaption and could be integrated with AFR to control the network delay and queuing delay together.

Cloud gaming. As a recently emerging application, cloud gaming also attracts the attention of researchers. Researches include the optimization of the renderer [23, 52, 27] and streaming codec [72], which are independent of the optimization at the transport level. There are also investigations on the user experience of cloud gaming [81, 71, 28], which could be integrated with our work by better customizing the optimization goal. Recent efforts also try to investigate the performance of production-level cloud gaming services from the client side [26], which are limited in scale and completeness. To the best of our knowledge, we are also the first piece of work to investigate the performance of production-level cloud gaming services from the server side, with the scale of tens of thousands of hours of playtime.

9 Conclusion

In this paper, we propose AFR to reduce the queuing delay of the decoder queue for high-quality RTC by dynamically adjusting the frame rate. AFR introduces a stationary controller and a transient controller to respectively mitigate the stationary heavy traffic and contingent arrivals and services. We further evaluate the performance of AFR with trace-driven simulations and deployments in the production clusters. Experiments demonstrate that AFR could significantly reduce the stuttering ratio and tail total delay.

This work does not raise any ethical issues.

Acknowledgements. We sincerely thank our shepherd KyoungSoo Park, anonymous reviewers, and labmates in the Routing Group from Tsinghua University for their valuable feedback. This work is sponsored by the National Natural Science Foundation of China (No. 62002196, 61832013, and 62221003) and the Tsinghua-Tencent Collaborative Grant. Bo Wang and Mingwei Xu are the corresponding authors.

References

[1] Cloud gaming (beta) with xbox game pass — xbox. https://www.xbox.com/en-US/xbox-game-pass/cloud-gaming, 2020.

[2] Peak signal-to-noise ratio - wikipedia. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio, 2020.

[3] Stadia - one place for all the ways we play. https://stadia.google.com/, 2020.

[4] Start - tencent cloud gaming. https://start.qq.com/, 2020.

[5] Your games. your devices. play anywhere — nvidia geforce now. https://www.nvidia.com/en-us/geforce-now/, 2020.

[6] Facebook 360 video. https://facebook360.fb.com/, 2021.

[7] Google meet and default video resolution - google meet community. https://support.google.com/meet/thread/58039897/google-meet-and-default-video-resolution, 2021.

[8] Huawei video conferencing platform — huawei enterprise. https://e.huawei.com/en/solutions/enterprise-collaboration/videoconferencing-platform, 2021.

[9] Meeting and phone statistics – zoom help center. https://support.zoom.us/hc/en-us/articles/202920719-Meeting-and-phone-statistics, 2021.

[10] Trtx 2080 ti vs rtx 3080 ti game performance benchmarks (i7-8700k vs core i9-10900k) - gpucheck united states / usa. https://www.gpucheck.com/compare/nvidia-geforce-rtx-2080-ti-vs-nvidia-geforce-rtx-3080-ti/, 2021.

[11] Vr-interactive – we are interactive. https://vr-interactive.at/, 2021.

[12] Youtube vr - home - youtube vr. https://vr.youtube.com/, 2021.

[13] Gfxbench - unified graphics benchmark based on dxbenchmark (directx) and glbenchmark (opengl es). https://gfxbench.com/result.jsp, 2022.

[14] multithreading - pin processor cpu isolation on windows - stack overflow. https://stackoverflow.com/questions/15324586/pin-processor-cpu-isolation-on-windows, 2022.

[15] Processor benchmarks - geekbench browser. https://browser.geekbench.com/processor-benchmarks/, 2022.

[16] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *Proc. USENIX NSDI*, 2018.

[17] Salahuddin Azad, Wei Song, and Dian Tjondronegoro. Bitrate modeling of scalable videos using quantization parameter, frame rate and spatial resolution. In *Proc. IEEE ICASSP*, pages 2334–2337, 2010.

[18] Richard Bellman and Robert Kalaba. On adaptive control processes. *IRE Transactions on Automatic Control*, 4(2):1–9, 1959.

[19] Ankita Bhutani and Preeti Wadhwani. Cloud gaming market share forecast 2025 — industry size report. https://www.gminsights.com/industry-analysis/cloud-gaming-market, 2020.

[20] Karl Bridge and Michael Satran. Multitasking - win32 apps — microsoft docs. https://docs.microsoft.com/en-us/windows/win32/procthread/multitasking, 2018.

[21] James Bruce, Marta Mrak, and Rajitha Weerakkody. Testing av1 and vvc - bbc r&d. https://www.bbc.co.uk/rd/blog/2019-05-av1-codec-streaming-processing-hevc-vvc, 2019.

[22] Alan Bryman and Duncan Cramer. *Quantitative data analysis with IBM SPSS 17, 18 & 19: A guide for social scientists*. Routledge, 2012.

[23] James Bulman and Peter Garraghan. A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines. In *Proc. USENIX HotCloud*, 2020.

[24] Ronald S. Bultje. The world's fastest vp9 decoder: ffvp9. https://blogs.gnome.org/rbultje/2014/02/22/the-worlds-fastest-vp9-decoder-ffvp9/, 2014.

[25] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Congestion control for web real-time communication. *IEEE/ACM Transactions on Networking*, 2017.

[26] Marc Carrascosa and Boris Bellalta. Cloud-gaming: Analysis of google stadia traffic. *arXiv:2009.09786*, 2020.

[27] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. T-gaming: A cost-efficient cloud gaming system at scale. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[28] Kuan-Ta Chen, Yu-Chun Chang, Hwai-Jung Hsu, De-Yu Chen, Chun-Ying Huang, and Cheng-Hsin Hsu. On the quality of service of cloud gaming systems. *IEEE Transactions on Multimedia*, 2013.

[29] Yushin Cho, William A Pearlman, and Amir Said. Low complexity resolution progressive image coding algorithm: progres (progressive resolution decompression). In *Proc. IEEE ICIP*, 2005.

[30] Tzu-Der Chuang, Pei-Kuei Tsung, Pin-Chih Lin, Lo-Mei Chang, Tsung-Chuan Ma, Yi-Hau Chen, and Liang-Gee Chen. A 59.5 mw scalable/multi-view video decoder chip for quad/3d full hdtv and video streaming applications. In *Proc. IEEE ISSCC*, pages 330–331, 2010.

[31] Harald Cramér. Mathematical methods of statistics, 1946. *Department of Mathematical SU*, 1946.

[32] Robert G. Gallager Dimitri P. Bertsekas. Section 3.3: The m/m/1 queuing system. In *Data Networks (2nd Edition)*, 1992.

[33] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *Proc. ACM SIGCOMM*, 2011.

[34] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *Proc. USENIX NSDI*, 2018.

[35] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. USENIX NSDI*, 2017.

[36] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. Abc: A simple explicit congestion controller for wireless networks. In *Proc. USENIX NSDI*, 2020.

[37] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proc. ACM SIGCOMM*. 2019.

[38] Jefferson Han and Brian Smith. Cu-seeme vr immersive desktop teleconferencing. In *Proc. ACM Multimedia*, pages 199–207, 1997.

[39] Refael Hassin and Moshe Haviv. *To queue or not to queue: Equilibrium behavior in queueing systems*, volume 59. Springer Science & Business Media, 2003.

[40] Petr Holub, Jiří Matela, Martin Pulec, and Martin Šrom. Ultragrid: low-latency high-quality video transmissions on commodity hardware. In *Proc. ACM Multimedia*, 2012.

[41] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. Gaminganywhere: an open cloud gaming system. In *Proc. ACM MMSys*, 2013.

[42] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proc. ACM SIGCOMM*, 2014.

[43] Zenja Ivkovic, Ian Stavness, Carl Gutwin, and Steven Sutcliffe. Quantifying and mitigating the negative effects of local latencies on aiming in 3d shooter games. In *Proc. ACM CHI*, pages 135–144, 2015.

[44] Van Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, 1988.

[45] JFC Kingman and MF Atiyah. The single server queue in heavy traffic. *Oper. Manage., Critical Perspect. Bus. Manage*, 2003.

[46] Marwan Krunz and Herman Hughes. A traffic for mpeg-coded vbr streams. In *Proc. ACM SIGMETRICS*, 1995.

[47] Ana Kuzmanic and Vlasta Zanchi. Hand shape classification using dtw and lcss as similarity measures for vision-based gesture recognition system. In *EUROCON 2007-The International Conference on" Computer as a Tool"*, pages 264–269. IEEE, 2007.

[48] Yun Gu Lee and Byung Cheol Song. An intra-frame rate control algorithm for ultralow delay h. 264/advanced video coding (avc). *IEEE Transactions on Circuits and Systems for Video Technology*, pages 747–752, 2009.

[49] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proc. ACM SIGCOMM*, 2020.

[50] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proc. ACM SIGCOMM*, 2019.

[51] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. Toward a practical perceptual video quality metric — netflix techblog. `https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652`, 2016.

[52] Xiaofei Liao, Li Lin, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. Liverender: A cloud gaming system based on compressed graphics streaming. *IEEE/ACM Transactions on Networking*, 2016.

[53] CC Lin, JI Guo, HC Chang, YC Yang, JW Chen, MC Tsai, and JS Wang. A 160kgate 4.5 kb skram h. 264 video decoder for hdtv applications. In *Proc. IEEE ISSCC*, pages 1596–1605, 2006.

[54] Candice Liu. Hardware decoding vs software decoding in 4k h264/h265 video. https://www.macxdvd.com/mac-video-converter-pro/hardware-decoding-4k-ultra-hd-video.htm, 2020.

[55] Andrea Lottarini, Alex Ramirez, Joel Coburn, Martha A Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler. vbench: Benchmarking video transcoding in the cloud. In *Proc. ASPLOS*, pages 797–809, 2018.

[56] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving Consistent Low Latency for Wireless Real Time Communications with the Shortest Control Loop. In *Proc. ACM SIGCOMM*, 2022.

[57] China Mobile and ZTE. Powered by sa: 5g mec-based cloud game innovation practice. GSMA 5G Case Studies (https://www.gsma.com/futurenetworks/wp-content/uploads/2020/03/Powered-by-SA-5G-MEC-Based-Cloud-Game-Innovation-Practice-.pdf), 2020.

[58] Omar Mossad, Khaled Diab, Ihab Amer, and Mohamed Hefeeda. Deepgame: Efficient video encoding for cloud gaming. In *Proc. ACM Multimedia*, 2021.

[59] Vit Niennattrakul and Chotirat Ann Ratanamahatana. On clustering multimedia time series data using k-means and dynamic time warping. In *2007 International Conference on Multimedia and Ubiquitous Engineering (MUE'07)*, pages 733–738. IEEE, 2007.

[60] Ilya Nikolaevskiy. Refactor framebuffer to store decoded frames history separately (i82be0eb3) · gerrit code review. https://webrtc-review.googlesource.com/c/src/+/116686, 2019.

[61] OPG609. List of 60fps games playable on ps5. https://www.reddit.com/r/PS5/comments/kiuh2t/list_of_60fps_games_playable_on_ps5/, 2020.

[62] Adrian Pennington. So you say you're planning a 16k live stream... - nab amplify. https://amplify.nabshow.com/articles/so-you-say-youre-planning-a-16k-live-stream/, 2022.

[63] Stefano Petrangeli, Viswanathan Swaminathan, Mohammad Hosseini, and Filip De Turck. An http/2-based adaptive streaming framework for 360 virtual reality videos. In *Proc. ACM Multimedia*, 2017.

[64] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. Improving mobile gaming performance through cooperative cpu-gpu thermal management. In *Proc. ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[65] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 1994.

[66] Elizabeth Ross, John Parente, Mike Jacobs, David Kuehn, John Baldwin, Corey Plett, Brock Mammen, and Liza Poggemeyer. typeperf — microsoft docs. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/typeperf, 2017.

[67] Saeed Shafiee Sabet, Steven Schmidt, Saman Zadtootaghaj, Babak Naderi, Carsten Griwodz, and Sebastian Möller. A latency compensation technique based on game characteristics to mitigate the influence of delay on cloud gaming quality of experience. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 15–25, 2020.

[68] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. IETF RFC 6298.

[69] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 2007.

[70] Arun Kumar Sharma. *Text book of correlations and regression*. Discovery Publishing House, 2005.

[71] Ivan Slivar, Lea Skorin-Kapov, and Mirko Suznjevic. Cloud gaming qoe models for deriving video encoding adaptation strategies. In *Proc. ACM MMSys*, 2016.

[72] Ivan Slivar, Mirko Suznjevic, and Lea Skorin-Kapov. The impact of video encoding parameters and game type on qoe for cloud gaming: A case study using the steam platform. In *Proc. IEEE International Conference on Quality of Multimedia Experience (QoMEX)*, 2015.

[73] James Stringer. Pushing it to the limit – parsec at 240 frames per second with approximately 4-8 milliseconds of ... — parsec. https://parsec.app/blog/parsec-game-streaming-total-latency-at-240-frames-per-second-c0818cc0daa5, 2022.

[74] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Supporting mobile vr in lte networks: How close are we? *Proc. ACM SIGMETRICS*, 2018.

[75] Tingfeng Wang, Zili Meng, Mingwei Xu, Rui Han, and Honghao Liu. Enabling high frame-rate uhd real-time communication with frame-skipping. In *Proc. ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2021.

[76] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 2004.

[77] Keith Winstein and Hari Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *Proc. USENIX ATC*, 2012.

[78] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proc. USENIX NSDI*, 2013.

[79] Jiyan Wu, Chau Yuen, Ngai-Man Cheung, Junliang Chen, and Chang Wen Chen. Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 2015.

[80] Gang Yi, Dan Yang, Abdelhak Bentaleb, Weihua Li, Yi Li, Kai Zheng, Jiangchuan Liu, Wei Tsang Ooi, and Yong Cui. The acm multimedia 2019 live video streaming grand challenge. In *Proc. ACM Multimedia*, pages 2622–2626, 2019.

[81] Saman Zadtootaghaj, Steven Schmidt, and Sebastian Möller. Modeling gaming qoe: Towards the impact of frame rate and bit rate on cloud gaming. In *Proc. IEEE International Conference on Quality of Multimedia Experience (QoMEX)*, 2018.

[82] Saman Zadtootaghaj, Steven Schmidt, Saeed Shafiee Sabet, Sebastian Möller, and Carsten Griwodz. Quality estimation models for gaming video streaming services using perceptual video quality dimensions. In *Proc. ACM Multimedia Systems Conference (MMSys)*, 2020.

[83] Yasir Zaki, Thomas Pötsch, Jay Chen, Lakshminarayanan Subramanian, and Carmelita Görg. Adaptive congestion control for unpredictable cellular networks. In *Proc. ACM SIGCOMM*, 2015.

[84] Huanhuan Zhang, Anfu Zhou, Jiamin Lu, Ruoxuan Ma, Yuhan Hu, Cong Li, Xinyu Zhang, Huadong Ma, and Xiaojiang Chen. Onrl: improving mobile video telephony via online reinforcement learning. In *Proc. ACM MobiCom*, 2020.

[85] Wenxiao Zhang, Feng Qian, Bo Han, and Pan Hui. Deepvista: 16k panoramic cinema on your mobile device. In *Proceedings of the Web Conference*, pages 2232–2244, 2021.

[86] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 2019.

[87] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. Learning to coordinate video codec with transport protocol for mobile video telephony. In *Proc. ACM MobiCom*, 2019.

[88] Chao Zhou, Mengbai Xiao, and Yao Liu. Clustile: Toward minimizing bandwidth in 360-degree video streaming. In *Proc. IEEE INFOCOM*, 2018.

[89] Dajiang Zhou, Shihao Wang, Heming Sun, Jianbin Zhou, Jiayi Zhu, Yijin Zhao, Jinjia Zhou, Shuping Zhang, Shinji Kimura, Takeshi Yoshimura, et al. An 8k h. 265/hevc video decoder chip with a new system pipeline design. *IEEE Journal of Solid-State Circuits*, 52(1):113–126, 2017.

[90] Dajiang Zhou, Jinjia Zhou, Xun He, Jiayi Zhu, Ji Kong, Peilin Liu, and Satoshi Goto. A 530 mpixels/s 4096x2160@ 60fps h. 264/avc high profile video decoder chip. *IEEE Journal of Solid-State Circuits*, 46(4):777–788, 2011.

[91] Dajiang Zhou, Jinjia Zhou, Jiayi Zhu, Peilin Liu, and Satoshi Goto. A 2gpixel/s h. 264/avc hp/mvc video decoder chip for super hi-vision and 3dtv/ftv applications. In *Proc. IEEE International Solid-State Circuits Conference*, pages 224–226, 2012.

Appendices

A Potential Solutions and Concerns

In this section, we discuss why other potential solutions are insufficient to address the problem in this paper, and discuss other concerns of adapting the frame rate during runtime.

A.1 Potential Solutions

Discarding frames or adjusting resolutions. For most widely adopted codecs, dropping one frame or changing the resolution will make the following frames fail to recover the raw pixels of the block because they are differentially encoded by the motion vector to the previous one[2]. This is to utilize the redundant information between frames to reduce the bitrate of the stream. Since key frames do not rely on previous frames, they are usually much larger than other predictive frames (sometimes 10×) [46]. Therefore, given the same bottleneck bandwidth, sending a frame with 10× larger size will take approximately 10× time (tens to hundreds of milliseconds), which drastically increases the delay for the users. Moreover, frequently requesting key frames will degrade the goodput of the streaming and potentially increase the congestion in the network. Therefore, directly dropping delayed frames at the client or frequently changing the resolution will introduce stalls for the subsequent frames and degrade the users' experiences of high-quality RTC.

Adjusting the bit-rate. Without changing the resolution and frame rate, adjusting the bit rate has a very limited effect in reducing the decoding delay. Generally speaking, resolution, bit rate, and frame rate could be independently set. The display resolution describes the number of distinct pixels in each dimension that

[2]Recent advances on scalable video coding could partially break the inter-frame dependency, yet degrades video quality with the same bit-rate [69].

can be displayed, and the frame rate represents the number of pictures within one second of video. And the bit rate represents the amount of data used for storing the coded bit-stream. So the higher resolution we set, the more pixels a single picture will have, which could mean a higher definition of the video. And setting a higher frame rate means there will be more pictures per video second to make the video smoother. If we set a higher target bit-rate while keeping other parameters unchanged, the encoder can use more data to represent the pictures to achieve lower possible image distortion with a lower *quantization parameter* [17].

In this case, with the unchanged frame rate and resolution, the decoding procedure is also unaffected. For example, in H.264/AVC, a sequence of macroblocks can be composed of a slice, a picture, therefore, is a collection of one or more slices. Slices are completely independent of each other, and the macroblocks inside a video frame can be reconstructed in parallel. The video decoding has been parallelized using slice-level or block-level parallelism. The resolution will affect how many pixels there are in one frame, and the frame rate determines the tolerable decoding delay for each frame. The parallelized decoder is not significantly affected by the precision of each pixel. We further measure the decoding performance with different bitrates in production in Appendix B.4.

Preset the frame rate and resolution based on client types. An alternative to AFR is that the application checks whether the hardware could reliably decode the video at a certain resolution and frame rate at initialization. This, however, would lead to underutilization on the client side. The decoding capability of hardware is fluctuating over time due to various reasons. For example, we measure the distribution of decoding delay of each user session in Appendix B.5. One-fourth of users will have at least 1‰ time of a long decoding delay of >18ms, which could result in severe queuing delay, as we illustrated in Figure 6. In this case, if we set the resolution and frame rate based on this tail metric, users will have a much lower resolution and frame rate during most of the time. Therefore, we need to control the frame rate in the runtime to dynamically adapt to the network and decoder dynamics.

Allocating the application with dedicated resources. Another seemingly feasible solution is to bind the application to a certain CPU core or GPU core to avoid the potential fluctuations caused by scheduling. However, we do not have such privileged control on client devices. As a user space application, the controllability over the user's system is limited. Even if an expert user pins the application to a certain core, for commercial systems such as Windows, pinning does not indicate isolating the core for that application only [14]. The system can only ensure the pinned application to run on that core, but could also schedule other processes if still available. Moreover, since our application is not CPU-intensive most of the time, there would usually be idle resources on the same core where the user binds the application to. Therefore, there could still be the same issue of latency increases *at tail*.

CPU	Release date	Score	Portion
Intel® Core™ i5-4590	Q2 2014	868	1.66%
Intel® Core™ i5-7200U	Q4 2016	481	1.61%
Intel® Core™ i5-9400F	Q1 2019	1058	1.56%
Intel® Core™ i5-4460	Q2 2014	801	1.41%
Intel® Core™ i5-5200U	Q4 2014	573	1.38%

Table 3: Top 5 CPU models of clients in our cloud gaming service.

GPU	Release date	Score	Portion
Intel® UHD Graphics 630	Q3 2017	888	4.54%
Intel® HD Graphics 4600	Q2 2013	474	3.42%
Nvidia GeForce GTX 1050Ti	Q4 2016	5059	3.19%
Intel® HD Graphics 630	Q3 2016	825	2.77%
Nvidia GeForce GT730	Q2 2014	863	2.48%

Table 4: Top 5 GPU models of clients in our cloud gaming service.

A.2 Practical Concerns

Since the frame-rate needs to be adjusted at the server, a straightforward concern is whether the frame-rate adaption over the Internet is timely for the stringent delay requirement of high-quality RTC. The measurements in production have two following findings. On one hand, the round-trip network delay is short enough to enable timely feedback: the average round-trip network delay is around 20ms of our cloud gaming service (Appendix B.2). Measurements over other high-quality RTC services (e.g., Google Stadia) have similar results of less than 20ms [26, 57]. On the other hand, the degradation of decoding delay usually lasts for a long time, with a median duration of more than 100 milliseconds (Appendix B.5). Moreover, we also demonstrate that the increase in decoding delay and network delay is hardly correlated (Appendix B.6). Therefore, for high-quality RTC, when the decoder fluctuates, it is timely enough to control the frame rate over the Internet.

B Measurement over Dataset

In this section, we supplement the observations in the main text with measurements in production. The measurement settings follow the details in §5.2.

B.1 User Characteristics

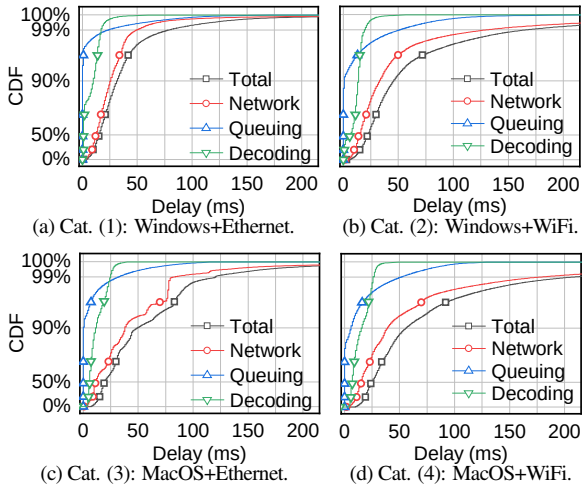
In addition to the distribution in §3.1, we present the top-5 models, with their release dates, benchmark scores, and portion in our users, of CPU and GPU in Table 3 and 4.

B.2 Delay Distributions

Compared to traditional RTC scenarios, the delay distribution for high-quality RTC has some unique features according to our measurements. We present the Cumulative distribution function (CDF) of component delays and the total delay to explore the delay patterns.

First, due to the edge deployments, the network delay in our cloud gaming service is quite small. According to Figure 20, the average round-trip network delay is approximately around 20ms. Even in this case, similar to traditional RTC services, the network delay is accounted for a large part of the total delay, the network delay line closely follows the total delay at the median for all four categories in Figure 20.

However, the tail delay of others component delays like



(a) Cat. (1): Windows+Ethernet. (b) Cat. (2): Windows+WiFi. (c) Cat. (3): MacOS+Ethernet. (d) Cat. (4): MacOS+WiFi.

Figure 20: Raw measurements of delays from production.

decoding delay and queuing delay are noticeable under cloud gaming scenarios. For the decoding delay, we can notice that the decoding delay for 1080p frames is 18ms at the 99th percentile. Note that the decoder of all sessions evaluated in this paper has been hardware-accelerated. Therefore, as analyzed in §3.1, the queuing delay is becoming noticeable at the tail. Referring to Figure 20, the 99th percentile of queuing delay can reach 50ms under categories (2) and (4), which could degrade users' experience for high-quality RTC services. We further present the root cause analysis below in Appendix B.3.

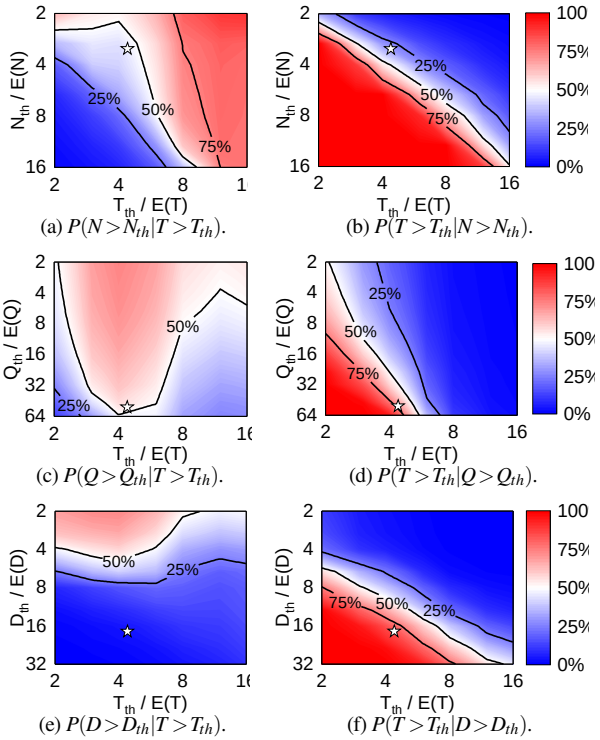
B.3 Root Cause Analysis

The total delay is mainly contributed by the network delay, decoding delay, and queuing delay §3. Therefore, we want to investigate how these three components contribute to the increase in total delay at the tail. For each frame, we denote T as total delay and C as component delay, where the component delay could be the network, decoding, or queuing delay.

To analyze the necessity and sufficiency of the component delay increasing to the total delay at the tail, we then calculate two conditional probabilities between the event of T longer than a certain threshold T_{th}, and the event of C longer than a certain threshold C_{th}:

- $P(C > C_{th}|T > T_{th})$. We want to account for how component delay increasing contributes to total delay under different delayed degrees T_{th}, and this conditional probability is subject to quantify it. If this conditional probability is close to one, there will be great confidence to blame the component delay for contributing C_{th} delay to the total delay to reaching T_{th}.
- $P(T > T_{th}|C > C_{th})$. As the sum of component delays, the total delay should increase when one of the component delays increases. This conditional probability is subject to illustrate this assumption and indicates the probability of total delay reaching the T_{th} under different component delay increasing degree C_{th}.

We calculate the conditional probabilities for three components for different C_{th} and T_{th}, and have the following observations.



(a) $P(N > N_{th}|T > T_{th})$. (b) $P(T > T_{th}|N > N_{th})$. (c) $P(Q > Q_{th}|T > T_{th})$. (d) $P(T > T_{th}|Q > Q_{th})$. (e) $P(D > D_{th}|T > T_{th})$. (f) $P(T > T_{th}|D > D_{th})$.

Figure 21: The heatmap of conditional probabilities for wired connections. The horizontal and vertical axes have been normalized by their average values. The star point's value is recorded in table 5 The down-left corner is 100% since the total delay should always be larger than the component delay.

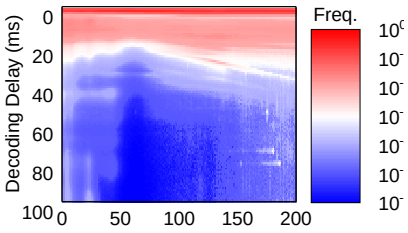
	Network	Queuing	Decoding
$P(C > C_{th}\|T > T_{th})$	44.7%	**56.6%**	4.0%
$P(T > T_{th}\|C > C_{th})$	29.8%	**69.5%**	84.2%

Table 5: Conditional probabilities with $T_{th} = 100ms$ and $C_{th} = 50ms$ for wired connections, which accounts for 82% of total users of our cloud gaming service.

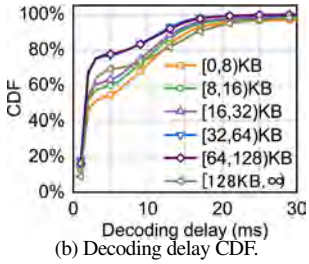
Total delay increasing is a reflection of components delay increases. As the sum of the different types of components delay, It's obvious that no matter what kind of component delay is increasing, the total delay will also increase.

So to find out the sufficiency of total delay increasing, we calculate the conditional probability of $P(T > T_{th}|C > C_{th})$ in right-side of Figure 21. We can notice that for all the component delays, their delay increasing can also mean a higher probability of total delay increasing (75%ile line in the figure is shifting to the right with the component delay increasing). The down-left corner is 100%, because as the sum of all types of component delay, the total delay must be larger than any component delay.

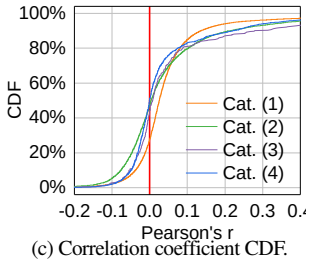
Queuing delay is responsible for delay increases of >100ms. To figure out the necessity of total delay increasing, we calculate the conditional probability of $P(C > C_{th}|T > T_{th})$ in left-side of Figure 21. Our major finding is that with the different order of severity of total delay increasing (2-16× of $\mathbb{E}(T)$, the root cause of it is also changing. As we can see, when T_{th} is larger than $8\mathbb{E}(T)$, network delay has a high probability



(a) Frequency heatmap.



(b) Decoding delay CDF.



(c) Correlation coefficient CDF.

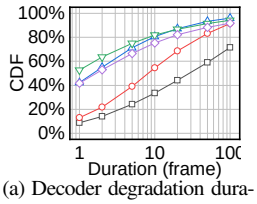
Figure 22: The correlation between the frame size and decoding delay for hardware decoders.

(shaded red) to be blamed. However, when T_{th} is from $3\mathbb{E}(T)$ to $8\mathbb{E}(T)$, queuing delay dominates the most increased events. It illustrates that the queuing delay is responsible for the increase of total delay by around 100ms. Specifically, we present the conditional probabilities for three components with $T_{th} = 100ms$ and $C_{th} = 50ms$ for wired connections in Table 5. As we can see, queuing delay has both high $P(C|T)$ and $P(T|C)$. Indicating that the total delay has a great possibility of reaching 100ms when queuing delay increases to 50ms. And for those video frames that total delay truly getting the 100ms, there will be great confidence to blame the queuing delay for contributing to the majority of delay increasing. So the queuing delay will be the root cause of the increase of total delay to 100ms.

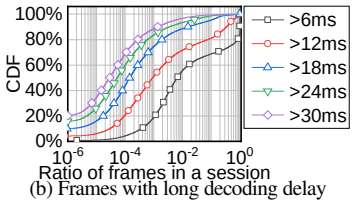
B.4 Decoding Performance

In this section, we explain the reasons behind the ineffectiveness of controlling the service process for eliminating queuing time by adjusting the bit rate. The decoding time of decoders mainly depends on the resolution of the streaming. However, due to the dependency between frames, changing the resolution during the streaming will make the subsequent frames undecodeable and needs to request a new key frame for most codecs [29]. Yet, since the frame size of key frames is usually several times ofthose of other frames [46], frequently requesting key frames will impose additional overhead on the network and degrade the users' experiences.

Another straightforward solution is to try to accelerate the service process by reducing the bit rate while maintaining the same resolution. With the same resolution and frame-rate options, reducing the bit rate means lesser video data per video frame can carry. We are to investigate whether sending video frames with smaller data sizes is helpful for decoding acceleration. However, according to our measurements on the H.264 decoder, merely changing the bit rate does not significantly reduce the decoding time.



(a) Decoder degradation duration.



(b) Frames with long decoding delay

Figure 23: Decoder degradation when filtered with different thresholds for decoding delay.

We measure the relationship between the frame size and decoding time of the dataset described in §5.2. We first present the heat map in Figure 22(a). With the variation of frame size, the distribution of decoding time does not significantly change, where the decoding time of most frame sizes intensively falls around several milliseconds, as shown in the red area at the top of the heat map. To eliminate the frame size variation under the same target bit rate, we split the frame size into different intervals and present the cumulative distribution function (CDF) in Figure 22(b). As the frame size become larger, the [128KB, ∞) the line does not locate in the rightest area (higher decoding delay). And other frame size interval's CDF lines stay together, indicating that the lowering frame size does not help for the decoding time acceleration.

Moreover, we split the dataset into four different categories (Table 1), to demonstrate that reducing frame size will not help decode acceleration under various platforms. We leverage the Pearson correlation coefficient to illustrate the independence, which value of zero can indicate that there is no association between the two variables [70]. Figure 22(c) shows that most of the Pearson's r value is located around zero, indicating the poor association between frame size and decoding delay. Therefore, controlling the service process of encoding bit-rate cannot effectively reduce the decoding time and alleviate the load of the decoder queue.

B.5 Decoder Degradation

Because the queue overhead will be introduced by the mismatch of the rate of two sides of the queue [39], if the decoding speed is not capable of processing the incoming default 60fps, it will be necessary for AFR to change to a lower target frame rate. However, since the client and server are located distant, the frame-rate adjustment request to the server side will need a control loop to take effect on the client side with the updated frame rate. So if the AFR control loop is shorter than the decoder degradation duration, the decoder will be capable of processing a higher incoming frame rate before the AFR requests take effect.

We measure the duration of the decoder degradation level over the traces introduced in §5.2. As we can see in Figure 23(a), for frames with a decoding time of more than 12ms, 50% of them last for more than 10 frames. Under 60fps streaming, considering the average of RTT is close to one frame interval of 16.7ms, and the 90%ile encoder response delay is less than three frames interval §6.4. In this case, lowering the frame rate will be helpful for alleviating the decoder queue even under the control loop delay of AFR. Therefore, AFR is capable of timely adjusting the frame rate to adapt to the decoder degradation. Moreover, the

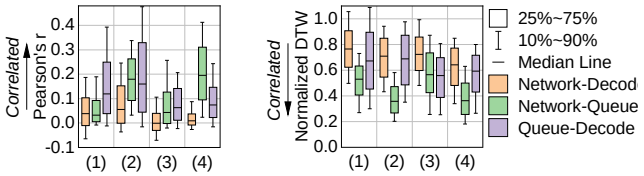
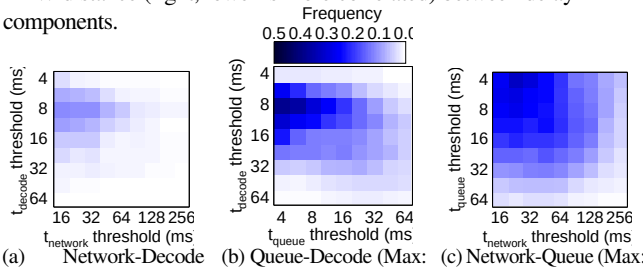


Figure 24: Pearson's r (left, higher is more correlated) and normalized DTW distance (right, lower is more correlated) between delay components.



(a) Network-Decode (Max: 0.12). (b) Queue-Decode (Max: 0.39). (c) Network-Queue (Max: 0.32).

Figure 25: Cramer's V between different delay components.

AFR can significantly help alleviate the queue overhead under those frames with a long period of decoder degradation and sustain queuing time for waiting for overhead queue elimination.

We further measure the ratio of frames with different decoding delays and present the results in Figure 23(b). Half of the user sessions suffer from a decoding delay of >12ms for at least 1‰ frames. This also indicates that the degradation of decoding delay is a general issue among all clients.

B.6 Component Correlation Analysis

The streaming pipeline will be affected by many components, like the networking, decoding, and queuing delays can both cause total delay increases to degenerate the user's experience Appendix B.3. In this paper, we propose AFR to reduce the tail queuing delay by matching the arrival rate of the decoder queue to the service rate (decoding speed). When decoding delay increases to disable decode frames timely, the AFR will send a frame-rate adjustment request from the client to the server. However, the request and subsequent frames need to be transported through the network. Therefore, a straightforward question is: does the increase of decoding delay affect the network delay to put an extra effect on the AFR control loop? We will figure out this by measuring the independence of those component delays.

We quantify the independence of different component delays with Pearson's r value [70], dynamic time warping (DTW) [18], and Cramer's v value [31]. In short, all these metrics demonstrate the poor association between networking and decoding delay, inclining that we could decouple the network and decoder issues and independently control them.

Regarding the Pearson correlation coefficient, the value of zero can indicate that there is no correlation between the two variables [70]. Figure 24 illustrates that for all four categories in Table 1, the Pearson's r value of networking and decoding are close to zero, indicating a poor correlation between them.

Moreover, the different component delays might be correlated with each other across frames. For example, the decoding delay

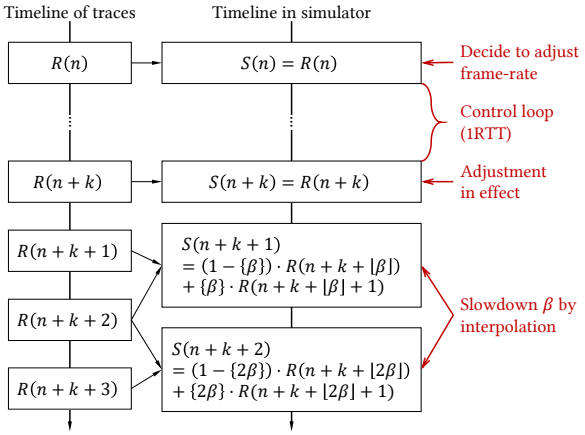


Figure 26: Illustration of frame-rate adjustment in our simulator.

could affect the subsequent queuing delay by its incapacity to decode video frames timely. To measure the correlation across frames, we leverage DTW to calculate the optimal match between two-time series [18]. The DTW algorithm is widely used in many scenarios like sign language recognition and time series clustering [47, 59]. The optimal match calculation under DTW is denoted by the match with minimal cost, where the cost is computed as the sum of absolute differences, for each matched pair of indices, between their values. Therefore, a larger DTW distance can be considered the mismatch between two series to a further extent. According to Figure 24, the normalized DTW distance of network delay to decoding delay under all four categories is large, showing the lack of correlation between them.

The strength of the relationship can also be assessed by Cramer's V value, which is a metric based on the χ^2-test but normalized for different data sizes. It indicates how strongly two categorical variables are associated [31]. A Cramer's V value of $\leqslant 0.1$ can be interpreted as hardly correlated [22]. According to our measurement in Figure 25, we can notice that all the Cramer's V values of networking and decoding delay are $\leqslant 0.2$, illustrating the weak association between networking and decoding state. Therefore, according to our measurements before, we can see the independence between networking and decoding delay.

C Simulator Implementation

In this section, we introduce the implementation of our simulator. Specifically, traces are recorded in the following format:

$$R(n) = \left[ts^{(n)}, \tau_{net}^{(n)}, \tau_{queue}^{(n)}, \tau_{decode}^{(n)}\right] \tag{9}$$

where $ts^{(n)}$ is the arrival timestamp of the n-th frame, τ_{net}, τ_{queue}, and τ_{decode} are the round-trip network delay, queuing delay, and decoding delay of that frame. The simulator reads the traces frame-by-frame at specific timestamps and measures the current frame rate based on the interarrival time as §4.2. The simulator then dequeues the head frame in the decoder queue when the decoder is available, where the decoding time of each frame is also read from the trace.

When the adaptive frame-rate decides to set the frame-rate

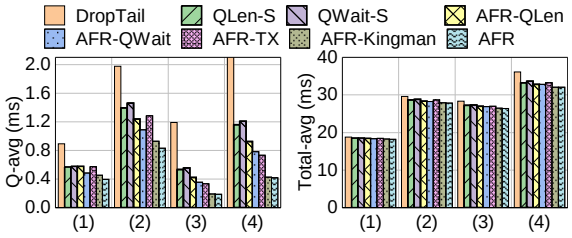


Figure 27: Average queuing delay (left) and total delay (right).

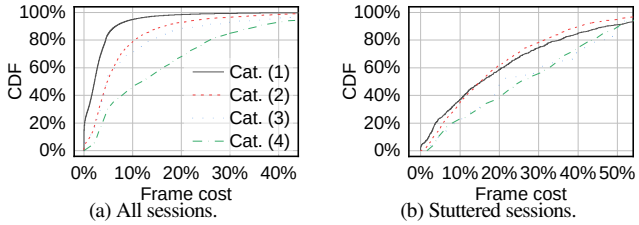


Figure 28: The number of wasted frames when skipping frames instead of adjusting the frame rate for AFR.

to f_{set}, the simulator first reads the current control loop by the round-trip network delay of the current frame $\tau_{net}^{(n)}$. The simulator then calculates the earliest frame $n+k$ that the new frame-rate f_{set} will take in effect:

$$k = \underset{k}{\operatorname{argmin}}\left(ts^{(n+k)} - ts^{(n)} \geqslant \tau_{net}^{(n)}\right) \tag{10}$$

After that, based on the measurement of the current frame-rate f_{cur}, the simulator calculates the slowdown factor $\beta = f_{cur}/f_{set}$, and reads the traces with a slowdowned speed. For example, as shown in Figure 26, When there are frames $R(n+k+1)$ to $R(n+k+3)$ in the original trace, the simulator reads the traces with indices $R(n+k+\beta), R(n+k+2\beta), \cdots$. When β is not integer, the simulator interpolates the traces with its neighbor frames ($S(n+k+1)$ and $S(n+k+2)$).

D Supplementary Experiments

D.1 Average Delay

We further measure the average queuing delay and total delay for four traces and present the results in Figure 27. As we can see, the reduction of tail delay of AFR does not sacrifice the average delay on all traces. In contrast, the average delay has also been slightly improved against baselines, due to the improvements at the tail.

D.2 Frame Costs of AFR with Skipping

Besides, as we discussed in §6.4, skipping frames without changing the frame rate from the content generator (e.g., gaming application) would waste the rendering resources of the server. For example, for high-quality RTC, rendering at 60fps would take approximately one time more GPU resources than rendering at 30fps. Therefore, we measure how many frames have been wasted (i.e., frame cost) if we merely skip the frames to approximate the target frame rate without adapting the content generator.

We present the results of all traces in Figure 28. For all traces, adjusting the frame rate could save 3% to 12% frame costs in all traces, saving considerable operating expenses for the service provider since GPU is one of the highest expenses. For stuttered sessions (following the definition in §6.2), the saved frame cost would be even higher.

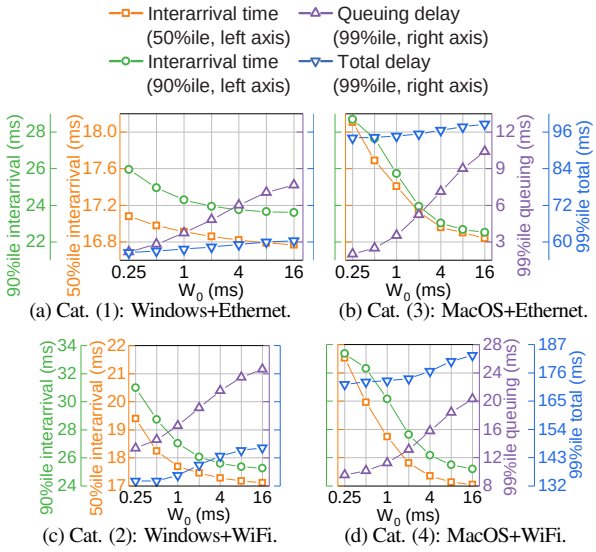


Figure 29: Sensitivity analysis on W_0 on different traces.

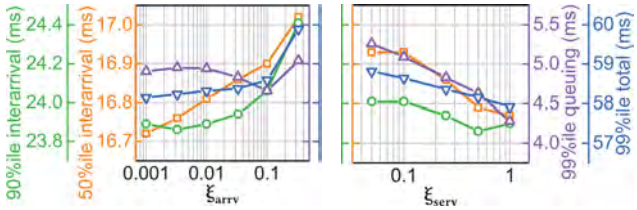


Figure 30: Performance of AFR with different settings of ξ_{arrv} and ξ_{serv}. Y-axes have been magnified compared to Figure 29.

D.3 Parameter Sensitivity

Long-term control target (W_0) We present the simulation results on the sensitivity of W_0 (in the stationary controller) on different traces in Figure 29. As we discussed in §5.2, a lower W_0 results in a more aggressive queue control yet leads to the degradation of frame rate. We vary W_0 from 0.25ms to 16ms and measure the interarrival time, queuing delay, and total delay. By adjusting W_0, operators could effectively balance the total delay and frame rate. Therefore, operators could adjust W_0 according to the preferences on total delay and frame rate for different users and games.

EWMA discounting factors (ξ_{arrv} and ξ_{serv}). We also vary the EWMA discounting factors (ξ_{arrv} for the arrival process and ξ_{serv} for the service process). Higher ξ indicates that the EWMA focuses on the recent values more to capture changes, while a lower value indicates more attention to the historical trends. As shown in Figure 30, the performance metrics (including the queuing delay, total delay, and frame rate) are relatively robust to these two parameters. By varying ξ_{arrv} and ξ_{serv} across several magnitudes, most metrics change marginally. For example, the 99%ile of queuing delay changes by 4× when varying W_0 (Figure 29) while only changes by less than 15% when varying ξ_{arrv} by three magnitudes (Figure 30). We also observe trends in varying ξ_{arrv} and ξ_{serv}. Lower ξ_{arrv} values will slightly

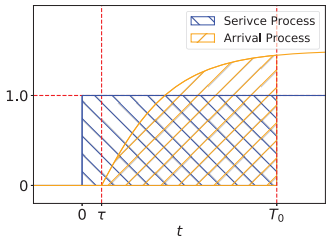


Figure 31: The system begins to control the queue after control-loop delay τ and stabilize the queue at T_0.

improve the performance of AFR, implying that the long-term behavior of arrival service is more critical. Higher ξ_{serv} also slightly improves the performance, indicating focusing on recent decoding time is helpful. This is because we have already filtered out outlier decoding time. Paying more attention to recent decoding time could make the AFR quickly adjust the frame rate.

E Convergence Analysis

Finally, we provide a detailed analysis of the convergence time during the state transitions of the stationary controller. As introduced in §4.2, let the expectation of queuing delay $E(\tau_{queue}) = W_0$, according to Eq. 1, we have:

$$\mu_a = \frac{\mu_s}{\rho} = \left(1 + \frac{c_a^2 + c_s^2}{2W_0}\mu_s\right)\mu_s \tag{11}$$

Then we can discuss the *convergence time* of the system. The convergence time here refers to the time at which the stationary controller converges to a stationary state when the service process changes, and the potential accumulated queue during the transition is drained up.

Specifically, without loss of generality, we discuss a simplified case shown in Figure 31: Both the arrival and service process have an average value of zero for $t < 0$, and the service process changes from zero to one at $t = 0$. The arrival rate will gradually respond to the change after a control loop of τ. We want to find the convergence time T_0 where

$$\int_0^{T_0} \mu_a \mathrm{d}t > \int_0^{T_0} \mu_s \mathrm{d}t \tag{12}$$

In this case, the queue accumulated during the response to the arrival rate will be cleared. We further illustrate the convergence in Figure 31. By substituting Eq. 11, we have:

$$\int_\tau^{T_0} \left(\mu_s + \frac{c_a^2 + c_s^2}{2W_0}\mu_s^2\right)\mathrm{d}t > \int_0^{T_0} 1 \mathrm{d}t \tag{13}$$

From the measurement of EWMA in Eq. 5, we have

$$\hat{\mu_s} = 1 - (1 - \xi_\mu)^{t-\tau} \quad (t > \tau) \tag{14}$$

Therefore, let $\gamma = 1 - \xi_\mu$ to simplify the expression, we need to find the minimum T_0 such that:

$$\int_0^{T_0-\tau} \left((1-\gamma^t) + \frac{c_a^2 + c_s^2}{2W_0}(1-\gamma^t)^2\right)\mathrm{d}t > T_0 \tag{15}$$

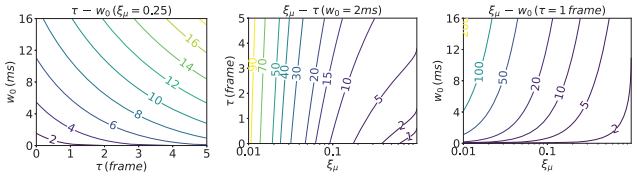


Figure 32: Contour plot of the convergence region of T_0 with different parameters.

By solving the integral in Eq. 15, finally we have

$$W_0 < \frac{c_a^2 + c_s^2}{2} \frac{(\gamma^{T_0-\tau} - 1)(\gamma^{T_0-\tau} - 3) + 2(T_0 - \tau)\ln\gamma}{2(\gamma^{T_0-\tau} - 1) + 2\tau\ln\gamma} \tag{16}$$

For example, when set $c_a^2 + c_s^2 = 2$, we vary the other parameters in Eq. 16 and present the minimum T_0 in Figure 32. In the most general settings of AFR ($\tau = 1$ since the average RTT is around 15ms, $\xi_\mu = 0.25$ as introduced in §5.2, $W_0 = 2$ms), the stationary controller can converge to the new stationary state within 2 frames. In other settings of the AFR parameters, the stationary controller could also converge and drain the queue within tens of frames, which is much less than the frame-rate adjustment interval of hundreds of frames as evaluated in §6.2.