Understanding and Taming the Inflated Latency in Mobile Cloud Rendering

YUANKANG ZHAO, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences, China
QINGHUA WU*, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences, Purple Mountain Laboratories, China
GERUI LV, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences, China
FURONG YANG, Institute of Computing Technology, Chinese Academy of Sciences, China
JIUHAI ZHANG, unaffiliated, China
FENG PENG, unaffiliated, China
YANMEI LIU, unaffiliated, China
ZHENYU LI, Institute of Computing Technology, Chinese Academy of Sciences, University of Chinese Academy of Sciences, Purple Mountain Laboratories, China
HONGYU GUO, unaffiliated, China
YING CHEN, unaffiliated, China
GAOGANG XIE*, Computer Network Information Center, Chinese Academy of Sciences, University of Chinese Academy of Sciences, China

Low-latency cloud rendering enables mobile users to experience high-quality, real-time 3D graphics but achieving low motion-to-photon (MTP) latency while maintaining smooth playback is a significant challenge. Our real-world measurement study identifies receive-to-composition (R2C) latency, caused by ineffective jitter buffer management, as the primary factor contributing to increased MTP latency. To address this, we introduce JitBright, a client-side optimization strategy that dynamically reduces MTP latency through adaptive jitter buffer management. By adjusting buffer levels based on smoothing

A preliminary shorter version [51] of this paper appeared at ACM NOSSDAV, 2024, where it received the Best Workshop Paper Award.

*Corresponding author: Qinghua Wu and Gaogang Xie.
This work was supported in part by the National Key R&D Program of China (2022YFB2901800).
Authors' addresses: Yuankang Zhao, Institute of Computing Technology, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, China; Qinghua Wu*, Institute of Computing Technology, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, and Purple Mountain Laboratories, China; Gerui Lv, Institute of Computing Technology, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, China; Furong Yang, Institute of Computing Technology, Chinese Academy of Sciences, China; Jiuhai Zhang, unaffiliated, China; Feng Peng, unaffiliated, China; Yanmei Liu, unaffiliated, China; Zhenyu Li, Institute of Computing Technology, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, and Purple Mountain Laboratories, China; Hongyu Guo, unaffiliated, China; Ying Chen, unaffiliated, China; Gaogang Xie*, Computer Network Information Center, Chinese Academy of Sciences, and University of Chinese Academy of Sciences, China.

ACM 1551-6865/2025/7-ART
https://doi.org/10.1145/3746283

playback probability and implementing proactive keyframe requests to mitigate frame dependency, JitBright minimizes both active and passive waiting times.

Our large-scale evaluation, conducted over 591,000 sessions across diverse network conditions (WiFi, 4G, 5G) and device types, demonstrates significant improvements in user experience. JitBright reduces median R2C latency by up to 87.5%, increases the proportion of sessions meeting strict MTP latency requirements by 6%-27%, and decreases the video freeze rate from 2.4%-2.8% to 0.4%-1.0%.

CCS Concepts: • **Networks** → **Network performance evaluation**; *Application layer protocols*.

Additional Key Words and Phrases: Mobile Cloud Rendering, Low Latency, Motion-to-photon Latency, Jitter Buffer

1 INTRODUCTION

Mobile cloud rendering is revolutionizing e-commerce by enabling real-time 3D graphics on resource-constrained mobile devices [39]. The ability to access high-quality, immersive visuals is reshaping how consumers interact with products online, enhancing engagement and driving business growth [3, 4, 35].

Cloud rendering transfers the processing burden from mobile devices to powerful servers, allowing even low-end devices to render complex scenes. This process introduces motion-to-photon (MTP) latency, which is the delay between user input (or motion) and the resulting image update (or photon) on a display [5]. The MTP latency must stay below 100 ms [21] or 150 ms [10] to maintain a seamless user experience. As shown in Figure 1, MTP latency is categorized into three components: *(i)* Rendering Engine (RE) latency, the time to render and encode a frame on the server; *(ii)* Network Transmission (NT) latency, the total time for uploading commands and downloading video frames; and *(iii)* Receive-To-Composition (R2C) latency, the duration from receiving the frame's last packet to its display.

Through large-scale online measurements from a leading e-commerce app in China (Section 4), we have identified R2C delay as the primary contributor to MTP latency. R2C delay arises from three processes: buffering, decoding, and rendering. Our in-depth analysis reveals that buffering latency accounts for the largest portion in the majority (71.5%) of cases, and this prolonged R2C latency persists across all grades of devices and network types. We further categorize the causes of buffering latency into two types: active waiting and passive waiting within the frame buffer (i.e., jitter buffer). Active waiting occurs when a frame is ready for decoding but is held to ensure smooth playback, while passive waiting happens when a frame must wait for the arrival of a reference frame required for decoding.

In this paper, we introduce JitBright, a systematic jitter buffer optimization strategy aimed at reducing Motion-to-Photon (MTP) latency by minimizing both active and passive waiting latencies across diverse network types and device grades. To achieve this goal, JitBright needs to address the following unique design challenges:

(i) Avoiding active waiting must balance the conflicting goals of low latency and smooth playout. The default strategy employed by WebRTC [31] clients is extremely conservative, resulting in unnecessary increases in active wait latency. Conversely, several studies pursue extremely low latency by minimizing the buffer level to zero [19, 24]. However, our evaluations in Section 6.2 indicate that this zero-buffering approach significantly degrades playout smoothness. To this end, JitBright introduces an adaptive gain to control the buffer level based on the probability that a frame will not play smoothly, thereby avoiding active waiting without compromising smoothness (Section 5.1).

(ii) Different devices and network types exhibit varying network and R2C latencies. As observed in Section 4.5, differences in device performance and network types affect the frame arrival process. Therefore, balancing latency and smoothness requires considering the variations in network types and device grades (Section 5.2).

(iii) Avoiding passive waiting by requesting a keyframe is promising, but at the expense of latency and smoothness. Passive waiting is caused by the decoding dependency between frames. An intuitive idea is to remove this dependency by proactively requesting a keyframe that can be decoded independently. However, this method may

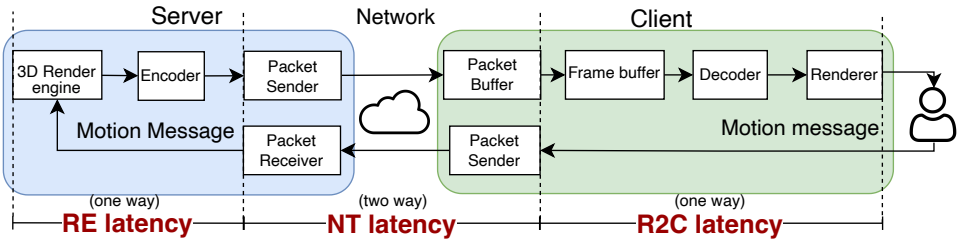


Fig. 1. Real-time cloud rendering architecture and latency decomposition.

introduce additional latency or trigger stalling due to the long transmission time of keyframes. As a solution, JitBright incorporates two cost functions to measure the cost of passive waiting or proactive requesting, and determines the action following the minimum cost (Section 5.3).

JitBright is designed to be lightweight and practical, making it easy to deploy. We have implemented JitBright in our real-world cloud rendering system (Section 5.5). Large-scale online A/B tests, conducted over more than 591,000 sessions, demonstrate that JitBright effectively reduces MTP latency while improving playout smoothness (Section 6). Specifically, JitBright reduces median R2C latency by 82.4%, 86.5%, and 87.5% for WiFi, 4G, and 5G, respectively. It also increases the proportion of sessions meeting the MTP latency requirement (i.e., less than 150 ms) by 15%–23%, 9%–20%, and 6%–27% on WiFi, 5G, and 4G networks, respectively. Furthermore, JitBright reduces the video freeze rate from 2.4%–2.8% to 0.4%–1.0%.

2 BACKGROUND

Cloud rendering system overview. Figure 2 depicts the architecture of our interactive cloud rendering system, which supports a cloud rendering service on a top e-commerce APP in China. The system incorporates a server and a mobile client. The server maintains the actual 3D models and renders them in real-time according to the user's motions (e.g., moving, changing viewpoints). Meanwhile, the server generates a video stream of the models and transmits it to the mobile client. In this way, the client can interact with lifelike 3D environments without suffering from the high computing costs [18]. Consequently, the user experience is impacted by MTP latency [36].

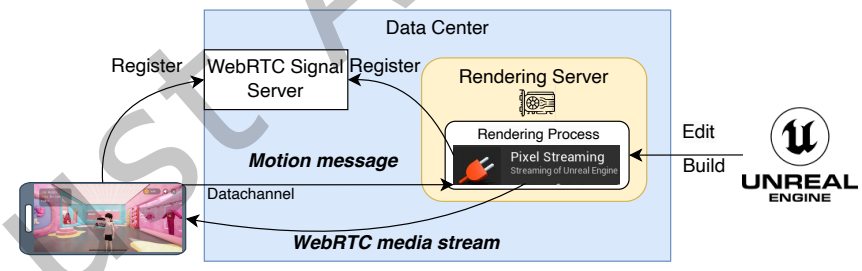


Fig. 2. Online cloud rendering system overview.

Cloud rendering communication workflow. The cloud rendering service operates on multiple rendering servers in 4 geographically distributed data centers in China. A separate WebRTC signal server is used to set up WebRTC connections with the client. Users access the cloud rendering service through a virtual shopping brand pavilion in the application on mobile clients. A session starts with a user entering the virtual pavilion in the APP. The mobile client first connects with the rendering server in the nearest data center. Then, it continually sends real-time user commands (e.g., motions like changing viewpoints) to the server through the WebRTC Datachannel utilizing SCTP [1], and receives real-time video streaming from the server by WebRTC.

Rendering engine. The cloud rendering service is built using Unreal Engine [14], a state-of-the-art 3D rendering engine famous for creating immersive 3D visualized environments. Each rendering service process

creates a series of real-time images of 3D models, which are captured into a live video stream using Pixel Streaming [16]. Pixel Streaming is a plugin within the Unreal Engine it contains a WebRTC module to encode and transport the images.

Video streaming with WebRTC. The rendered image stream is encoded and transmitted with the WebRTC [31]. We choose WebRTC because it already serves as a standard interactive video streaming framework [20] and is widely supported by major web browsers and platforms [42], thus facilitating large-scale deployments.

Video encoding. In the encoding process of video streaming, two types of frames exist: keyframe (I-frame) and delta frame (P-frame). While each keyframe can be decoded as an independent image, each delta frame must be decoded based on its reference frame, which is a previous keyframe or delta frame. A keyframe is generally 4-10 times larger than a delta frame [37] because it contains the full image data required for independent decoding. In contrast, delta frames only store the differences between consecutive frames.

3 JITBRIGHT OVERVIEW

We introduce JitBright, a client-side jitter buffer optimization algorithm designed to reduce Motion-to-Photon (MTP) latency in mobile cloud rendering applications. Figure 3 shows JitBright's components. JitBright comprises three key components: latency diagnostic, jitter delay manager, and keyframe requester, which together monitor and control the jitter buffer strategy on the client side in real-time.

Latency diagnostic (Section 4). Latency diagnostic identifies the bottlenecks in MTP latency. By a data-driven analysis of MTP latency analysis, we pinpoint the major contributors to MTP latency, such as active and passive waiting times in the jitter buffer. These latencies primarily stem from inefficient buffer scheduling and frame dependencies during the decoding process. It also adjusts the control parameters of the jitter delay manager based on network type and device grade, ensuring optimal performance across various conditions.

Jitter delay manager. The jitter delay manager dynamically adjusts the buffer level using an adaptive gain mechanism (Section 5.1). This ensures that the buffer level is optimally managed based on the likelihood of smooth playback, balancing low latency and playback smoothness. The adaptive gain's smooth parameter is adjusted according to the device and network type provided by the latency diagnostic (Section 5.2).

Keyframe requester (Section 5.3). The keyframe requester proactively reduces passive waiting by requesting keyframes at critical times, mitigating latency due to frame dependencies.

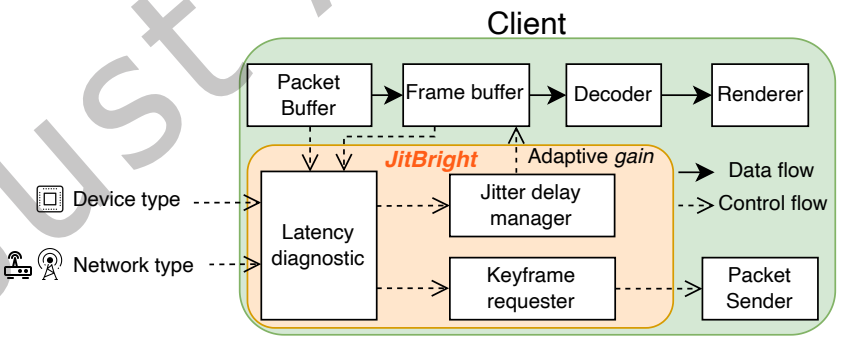


Fig. 3. An overview of JitBright

4 MTP LATENCY DIAGNOSTICS

In this section, we introduce how JitBright diagnoses the MTP latency. For this purpose, we address four issues:

(i) How large is MTP latency in mobile cloud rendering? We use measurements to quantify in Section 4.2. Our detailed measurement identified that R2C latency is the main factor that inflates MTP latency. And this phenomenon is ubiquitous across all device types and network types.

(ii) What are the primary factors that inflate R2C latency? We break down R2C latency into three elements: buffering, decoding, and rendering latency, and find the root cause is the active waiting and passive waiting in the buffering process.

(iii) Why are the active and passive waiting times so prolonged? We identified that an inefficient jitter buffer scheduling strategy is primarily to blame.

(iv) How do network type and device type affect MTP latency? We found that different device types and network types exhibit varying tail network latency and decoding latency. This suggests that different scenarios necessitate distinct jitter buffer management configurations.

4.1 Measurement Setup

Methodology. The cloud rendering system periodically (i.e., every 30 seconds) performs end-to-end online measurements to record the latency of each component. Recalling Figure 1, in each measurement, the client first sends a measurement message to the server. Once the server receives the message (the first part of *NT Latency*), it will provide feedback to the client regarding the rendering and encoding time of the latest frame, corresponding to *RE Latency*. After that, the server transmits the video frame to the client (the second part of *NT Latency*). The client stores the received frame in its frame buffer, and then decodes and renders it at a specific time, which composes *R2C Latency*. The client calculates latencies of each component using feedback from the server.

The latency of each component is obtained by modifying related callback functions on both sides. Additionally, the user's device type and network access type are also collected. Once the client record is generated, it is immediately sent to the server for analysis. Although each measurement involves multiple frames, only the first one's MTP latency is recorded. This is because generating, storing, and transmitting the record of each frame is costly in the online system.

Dataset. The data collection spanned 18 days, from July 2 to 19, 2023. It encompassed 36 million frames, accumulated a total video time of 166 hours, and involved the participation of over 2,000 users. Note that only performance-related information was collected during the anonymous user's access, which does not raise any ethical issues.

4.2 Analyzing MTP Latency in Mobile Cloud Rendering

We conduct a detailed analysis of MTP latency in mobile cloud rendering, focusing on its key components and how they vary across different device categories.

We first investigate the latency distribution of the three components that compose MTP latency, i.e., RE latency, NT latency, and R2C latency. The results in Figure 4 indicate that: *(i)* RE latency is relatively stable, with an average value of 30 ms per frame. *(ii)* NT latency exhibits a long-tailed distribution, as reported in previous studies [22, 23, 46]. *(iii)* **R2C latency accounts for the highest proportion of MTP latency in most (57.2%) cases.** These results imply that MTP latency is primarily affected by R2C latency.

To confirm this conclusion, we further investigate the relationship between R2C latency and MTP latency. We calculate the *conditional probability distribution* between R2C latency and MTP latency, namely $P(T_{R2C} > t_y | T_{MTP} > t_x)$. Here, T_{R2C} and T_{MTP} denote R2C and MTP latency, respectively. t_x and t_y are latency values, corresponding to the horizontal and vertical coordinates in Figure 5, respectively. The results show that when MTP latency exceeds the latency requirements (i.e., over 150 ms), over 50% of cases experience R2C latency over 100 ms (i.e., 2/3 of 150 ms).

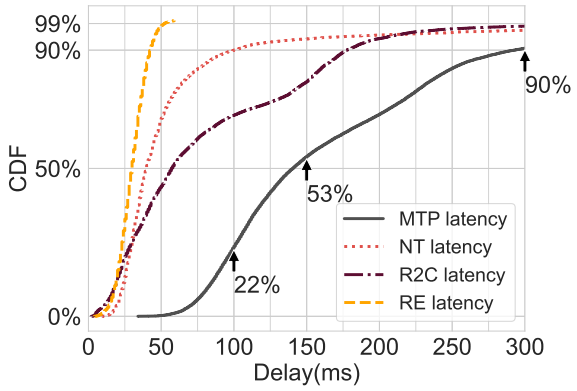


Fig. 4. CDF of MTP latency and its three components.

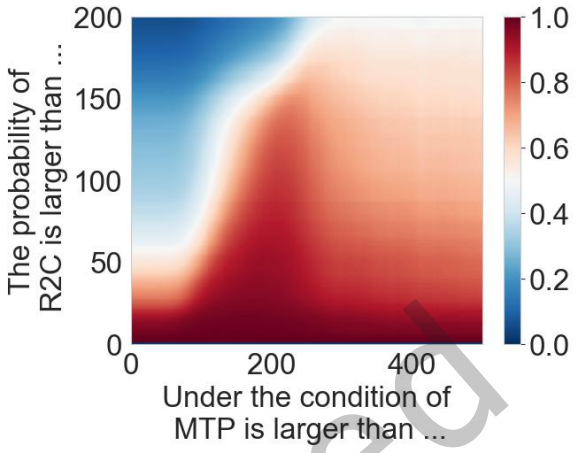
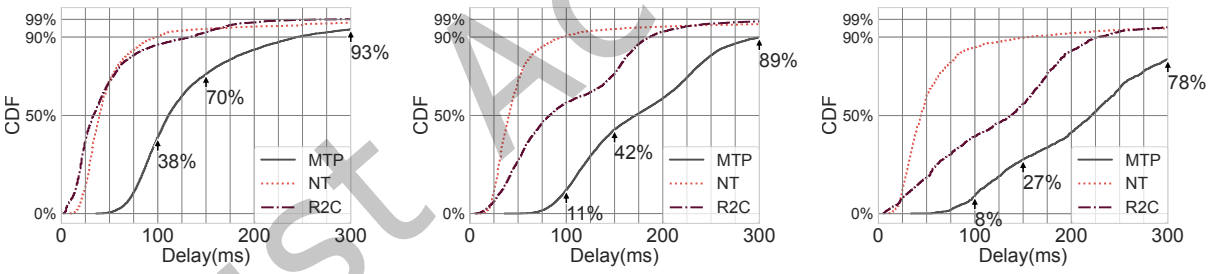


Fig. 5. The conditional probability of R2C latency and MTP latency.

We further analyze the issue and find that it exists across all device grades. The devices were categorized into high, medium, and low grades based on their CPU performance and benchmark scores [17]. The distribution of device grades is shown in Table 1. We recorded MTP latency, network latency, and R2C latency for high-end, medium-end, and low-end devices, respectively. RE latency is omitted, as we observed it to be stable. Figure 6 presents their distributions, where it can be observed that for high-end devices, the R2C latency is as substantial as the network latency, with both showing nearly identical distributions. Furthermore, for medium-end and low-end devices, R2C latency constitutes the largest portion of MTP latency, this leads to the following observation:



(a) High-end (b) Mid-end (c) Low-end

Fig. 6. CDF of MTP latency and its components. Prolonged R2C latency is present across all device grades.

***Observation*: Unsatisfactory MTP latency is dominated by inflated R2C latency, and this phenomenon is ubiquitous across all types of devices.**

4.3 Key Factors Inflating R2C Latency

We then explore the underlying factors contributing to the inflation of R2C latency, breaking down the processes and mechanisms that lead to this bottleneck in mobile cloud rendering.

As we have identified R2C latency is the bottleneck in optimizing M2P latency, the next question is: *What are the primary factors that inflate R2C latency?*

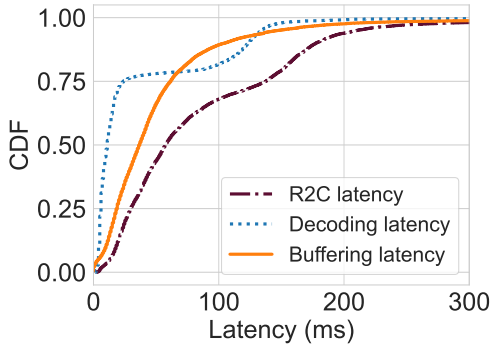


Fig. 7. CDF of R2C latency and its two components.

Decomposing R2C latency. R2C latency corresponds to the duration from the client receiving a video frame to the user seeing this frame played. Recalling Figure 1, the client establishes a frame buffer, known as the *jitter buffer*, to store each received frame in timestamp order. The decoder fetches the earliest frame from the jitter buffer, decodes it, and finally passes it to the renderer for display. Hence, R2C latency is produced in three processes: *(i)* Buffering, *(ii)* Decoding, and *(iii)* Rendering. Figure 7 shows the latency distribution of each process. Note that rendering latency is omitted because it is always less than 10ms [8]. The results show that **buffering latency is the key factor impacting R2C latency in most (71.5%) cases.**

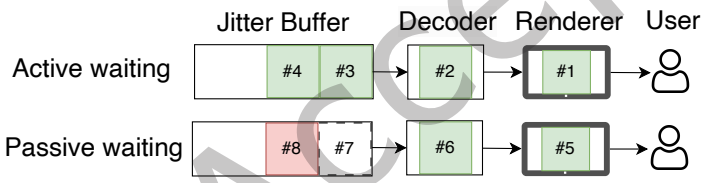


Fig. 8. Active and passive waiting in the jitter buffer.

Understanding buffering latency. Buffering latency indicates that a frame is waiting in the jitter buffer rather than being decoded. It is caused by two cases: *active waiting* and *passive waiting*, as illustrated in Figure 8. Active waiting occurs when frames are queued in the jitter buffer, awaiting their scheduled decoding(such as frames #3 and #4). On the other hand, passive waiting occurs when frames arrive out of order, and later frames (frame #8) cannot be decoded because their previous reference frames (frame #7) have not yet arrived. This phenomenon can be viewed as head-of-line blocking of frames [11], and is usually caused by packet loss in the transmission [33]. Recall that the delta frame (P-frame) must be decoded based on its reference frame, which is a previous keyframe or delta frame.

Therefore, the root cause of the inflated R2C latency is:

***Root cause*: Active and passive waiting in the client jitter buffer primarily inflates R2C latency.**

4.4 Ineffective Jitter Buffer Scheduling Strategy

To analyze active and passive waiting behavior on the client side, we perform controlled experiments on our local testbed (Section 5.5), for the ease of accessing fine-grained latency information. The results (stationary, connected with WiFi) indicate that the latency brought out by active waiting accounts for 72.8% of R2C latency. In this case, multiple successive video frames are queued in the jitter buffer, waiting to be decoded. In other words, R2C latency is primarily caused by the *ineffective jitter buffer scheduling strategy* [13]. Jitter buffer aims to enable the smooth playout of video frames. Specifically, when the bandwidth suddenly drops or a frame is too large, causing

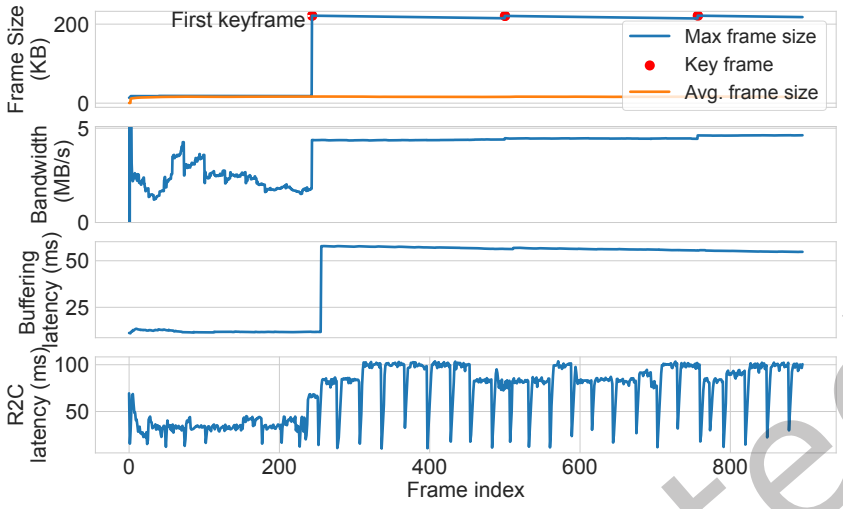


Fig. 9. The impact of maximum frame size and its frequency on the estimated value of jitter delay.

the transmission time to exceed the inter-frame interval (e.g., 16.7ms at 60 fps frame rate), the frames in the jitter buffer can be played to avoid stalling or stutter.

Default strategy. The default scheduling strategy of the jitter buffer in the WebRTC-based cloud rending system follows an intuitive idea: the buffer level (denoted as B) should be higher when the frame size (denoted as L) is fluctuating or the link bandwidth (denoted as C) is insufficient, leading to the following:

$$B \propto \frac{L_{max} - L_{avg}}{\hat{C}}, \tag{1}$$

where $\propto$ indicates "proportional to". L_{max} and L_{avg} are the smoothed maximum and average frame sizes (details in [52]), respectively. $\hat{C}$ is the estimated bandwidth, using Kalman Filter [6]. It is worth noting that L_{max} is *not* a fixed constant: WebRTC updates it for every decoded frame. Let $L(t_i)$ be the current frame size and $\psi \in (0, 1)$ the reduction factor (default $\psi = 0.9999$ [2]). If $L(t_i) > \psi \cdot L_{max}$ the buffer treats the frame as "large" and sets $L_{max} \leftarrow L(t_i)$. Otherwise, L_{max} decays exponentially as $L_{max} \leftarrow \psi \cdot L_{max}$. Figure 9 illustrates this behavior, showing L_{max} rising sharply on a keyframe and then decreasing slowly as only delta frames follow. This scheme provides a moving upper bound for Equation (1).

Performance issue. In video streaming, the frame size is determined by both the frame type (keyframe or delta frame) and the content complexity [32] when the target encoding bitrate is constant. In particular, a keyframe is generally 4-10 times larger than a delta frame [37], dominating the variation in frame size. Therefore, L_{max} in Equation 1 always indicates the size of a keyframe. The default strategy tends to maintain a larger number of frames in the jitter buffer to avoid the smoothness affected by keyframes. However, keyframes appear infrequently in the cloud rendering system, with typically one keyframe every 300 frames, following the common practice [15, 19]. Under this circumstance, the default strategy is too conservative, resulting in unnecessary active waiting and ultimately inflated R2C latency.

Figure 9 gives an example from experiments in our controlled testbed. The client stays stationary and connects with the server through an Ethernet cable. The frame rate is set to 60 fps. It can be observed that the buffering latency suddenly increases from 12 ms to 57 ms (4.8x) after the first keyframe appears, and remains at a high level (over 52 ms) thereafter. R2C latency exhibits a similar behavior, with an increase from 25 ms to 80 ms (3.2x).

4.5 Understanding the Impact of Network Type and Device Type on MTP latency

Our further research revealed that both network access type and device type affect the network latency and R2C latency. Therefore, the *Latency Diagnostics* module adjusts JitBright control parameters based on the current device and network types to address the characteristics of different scenarios.

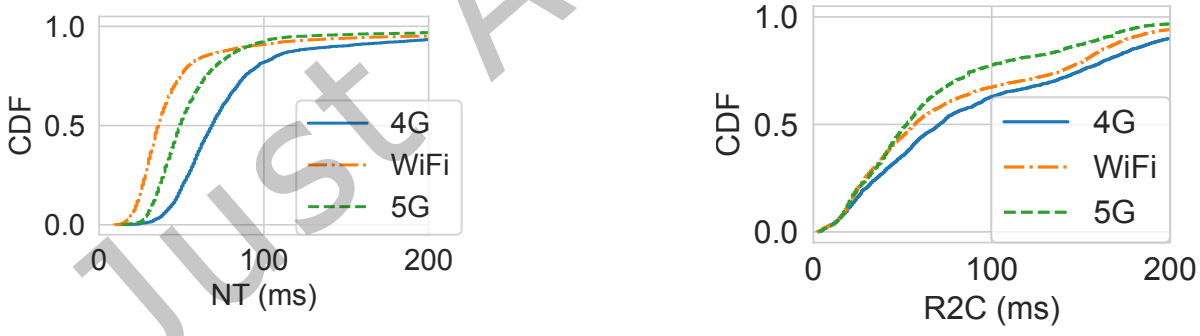
Network type. The network latency differs across various network types, with 4G having the highest median latency (69 ms), followed by 5G (49 ms), and WiFi with the lowest (36 ms), as shown in Figure 10a, which aligns with existing studies [27, 44]. Such latency variation affects the frame arrival process [38, 47] and consequently impacts the R2C latency, as illustrated in Figure 10b. Specifically, the median R2C latency for WiFi (57ms) and 5G (52ms) are relatively close, whereas 4G exhibits a higher median of 71 ms. Additionally, the 4G's P90 tail latency is 200 ms, which is 13% higher than WiFi and 23% higher than 5G.

Interestingly, the median R2C latency for WiFi is not lower than that of 5G despite its lower NT, a seemingly counter-intuitive result that two factors can explain. First, users on 5G typically operate newer devices equipped with hardware decoders capable of faster frame decoding: Figure 11a shows that 90.6% of 5G frames finish decoding within 100ms, whereas the ratio for WiFi is 81.1%. Moreover, WiFi users constitute a larger proportion of the user base and include more mid to low-end devices (see Table 1), resulting in worse tail decoding performance.

Second, WiFi links exhibit higher short-range jitter. Specifically, for sessions where NT < 100 ms, we separately analyze the 0–50 ms and 50–100 ms ranges, since approximately 65% of net latency samples are below 50 ms, and nearly 90% are below 100 ms, reflecting typical RTT values observed under normal network conditions. In both ranges, the standard deviation of NT is larger for WiFi than for 5G (Figure 11b), consistent with findings from previous studies [44]. Consequently, WebRTC must maintain deeper jitter buffers to handle these bursty variations, offsetting WiFi's median NT advantage and resulting in slightly higher overall R2C latency.

Finally, as confirmed by the nearly identical downlink *bitrate* distributions for WiFi and 5G sessions (fig. 11c), we rule out the potential confounding effect of larger keyframes in WiFi scenarios. These findings lead to the following implication:

Implication 2: *Different jitter buffer management strategies are required due to variations in the frame arrival process under different network types.*

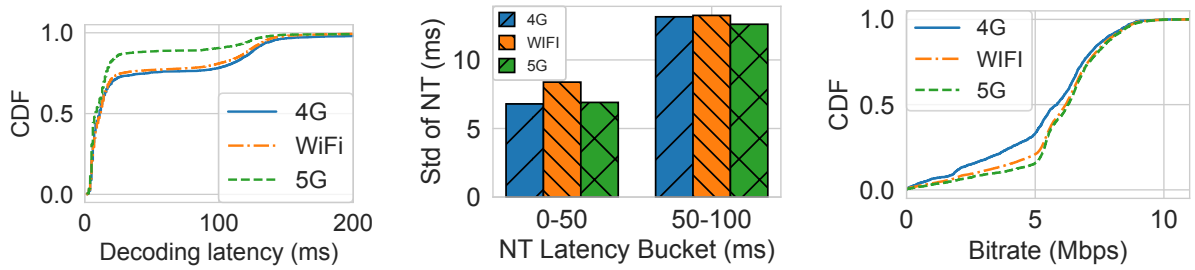


(a) Network latency under different network types

(b) R2C latency under different network types

Fig. 10. Network latency and R2C latency under different network access types.

Decive type. We recorded the network latency for different device types. The distribution of network types is as shown in Table 1. Users access this application via wireless networks, mainly WiFi, which accounts for 77.2% of all samples. As shown in Table 2, the P50 network latency differs by only 10 ms between high-end and low-end devices. By contrast, the gap widens sharply in the tail: at the 99th percentile, the latency of mid-end devices is



(a) Decoding latency CDF (b) Short-range NT jitter (c) Bitrate CDF

Fig. 11. Comparison of device-side decoding latency, short-range network jitter, and downlink bitrate for 4G, WiFi, and 5G sessions.

Table 1. Proportion of device grades and network type.

	Device Grade		
Network Type	High(37.8%)	Mid(40.0%)	Low(19.7%)
WiFi(77.2%)	26.9%	32.5%	17.8%
4G(10.7%)	4.7%	4.1%	1.9%
5G(9.6%)	6.2%	3.4%	0.02%

Table 2. Network latency at different percentiles for three device type

	P10	P50	P99
high-end	22.8 ms	36.0 ms	1813.8 ms
mid-end	24.8 ms	40.0 ms	2691.2 ms
low-end	26.8 ms	46.0 ms	3150.0 ms

1.48 times that of high-end devices, and low-end devices reach 1.7 times. For clarity, our *network latency (NT)* metric includes both RTT and frame transmission time. At the 99th percentile, NT significantly exceeds RTT (348ms) because the frame transmission time contributes a larger proportion compared to RTT, primarily due to many large keyframes at the tail (e.g., 1813.8ms for high-end devices in Table 2). In conclusion, the impact of device grade on network latency reveals that under non-tail conditions, network latency for high-end, mid-end, and low-end devices remains similar. However, under tail conditions, the differences in network latency across device grades become significantly more pronounced. This leads to the following implication:

Implication 1: *A more conservative jitter buffer management strategy is required to accommodate the worse tail latency conditions in lower-end devices.*

In summary, our large-scale measurements of the online cloud rendering system reveal that optimizing MTP latency is bottlenecked by R2C latency. However, R2C latency is significantly affected by the jitter buffer scheduling strategy on the client side. This motivates us to explore an effective strategy that performs well in the wild mobile Internet, accommodating diverse network types and device types.

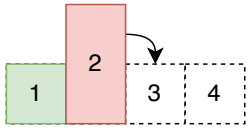


Fig. 12. A case of an oversized frame can not be transmitted within an extra frame interval.

5 DEFLATING MTP LATENCY BY JITBRIGHT

Minimizing MTP latency is essential for delivering a seamless user experience in cloud rendering. As discussed in Section 4, the key to reducing MTP latency lies in optimizing the jitter buffer scheduling strategy. The ideal approach must balance two conflicting objectives: *(i) reducing latency*, to prevent MTP latency from exceeding acceptable limits (e.g., over 150 ms); and *(ii) ensuring smooth playback*, to avoid inconsistent frame rates or stalling.

To address this challenge, we propose JitBright, a systematic jitter buffer optimization strategy that deflates MTP latency while ensuring smooth playout. Specifically, JitBright introduces a jitter delay manager and a proactive keyframe requester. The jitter delay manager sets an adaptive gain to control the jitter buffer level based on the probability that a frame will not play smoothly, thereby avoiding unnecessary active waiting. The keyframe requester proactively sends keyframe request to remove the decoding dependency between frames, thus reducing passive waiting latency.

5.1 Jitter Delay Manager with Adaptive Gain

To balance the requirements of low latency during active waiting and ensure smooth playout, JitBright introduces a jitter delay manager that adaptively adjusts a *gain* parameter applied to the default strategy. Specifically, it transforms Equation 1 as follows:

$$B = gain * \frac{(L_{max} - L_{avg})}{\hat{C}}. \tag{2}$$

Design principle. In principle, the *gain* should be proportional to *the probability of a frame not satisfying the smoothness requirement*. That occurs when a frame exceeds the average size L_{avg}, and the extra part cannot be transmitted within a single frame interval (denoted as I) as shown in Figure 12. Consequently, this frame will arrive later than expected, resulting in uneven playout speed or a stall. The principle of setting *gain* is formulated as:

$$gain \propto P(L - E(L) \geq S), \tag{3}$$

where L represents the random variable of frame size, and $E(L)$ means expected frame size, corresponding to L_{avg}. $P(\cdot)$ indicates the probability. S is the amount of data that can be transmitted within the frame interval I. Note that I is constant and directly derived from the encoding frame rate. In our setting (Section 6), I is 16.7ms for 60 fps video streaming. To further control the preferences for latency or smoothness, we introduce a control parameter sp (smooth parameter) in S, as presented in Equation 4. sp is set according to network and device types as discussed in Section 5.2.

$$S = sp \times I \times \hat{C}. \tag{4}$$

Based on *Chebyshev's inequality* [26], the upper bound on the probability of violating the smoothness requirement is illustrated by Equation 5.

$$P(L - E(L) \geq S) < P(|L - E(L)| \geq S) \leq Var(L)/S^2. \tag{5}$$

To this end, we define the *gain* considering the upper-bound probability of a smoothness violation event, as in Equation 6.

$$gain = Var(L)/S^2. \tag{6}$$

The gain in Equation (6) scales the jitter delay estimated by WebRTC's built-in jitter estimator [13], which already captures short-term network-delay variance. Thus, the gain corrects only the remaining dominant factor, namely the frame-size variability identified in Section 4.3. Online measurements support this distinction: 99.8% of packets affected by network jitter experience less than one frame interval (16 ms) of additional delay, indicating that the impact of network-induced jitter is negligible relative to the effect of frame-size variation.

Control logic. Smoothness is ensured when successive video frames arrive at the client at a uniform speed. This occurs when the variation in frame size (i.e., $Var(L)$) is small or when the bandwidth is sufficient (i.e., S is large). In this case, the *gain* is set to a small value, leading to a low buffer level as well as low latency. Otherwise, the *gain* will increase adaptively to reserve more frames in the jitter buffer, achieving smooth playout. Due to the relatively wide upper bound of the Chebyshev inequality, this algorithm initially favors smoothness. In latency-sensitive scenarios, the smoothness parameter sp can be adjusted to meet the requirements, as described in Section 6.

There is a special case for this algorithm: when the client requests a keyframe (see Section 5.3), the *gain* is set to 1 and maintained until the keyframe is received. This is because a large frame is expected to arrive, and the client jitter buffer should be prepared for that.

5.2 Adaptive Management of Device and Network Diversity

Different network and device types influence both network latency and R2C latency, leading to variations in frame arrival processes, as discussed in Section 4.5. Consequently, the jitter delay manager should consider the specific characteristics of network types and device types. Specifically, we control the smoothness parameter of adaptive gain based on different network and device conditions. Formally, this is expressed as:

$$sp^* = \arg\min_{sp} \mathcal{L}(T(sp \mid d, n), S(sp \mid d, n)) \tag{7}$$

where, sp^* is the optimal control parameter for the given scenario. $\mathcal{L}(T(sp \mid d, n), S(sp \mid d, n)$ depends on the average R2C latency T and the average stutter rate S , both of which are influenced by the smoothness parameter sp , as well as the device type d and the network type n . The optimization seeks to find the sp^* that maximizes the quality function under the given device and network conditions.

We define $\mathcal{L}(T, S)$ as:

$$\mathcal{L}(T, S) = \frac{T}{T_{max}} + \frac{S}{S_{max}} \tag{8}$$

where T is the average R2C latency, normalized by T_{max}, the maximum tolerable R2C latency (default value is 116 ms); S is the average stutter rate, normalized by S_{max}, the maximum tolerable stutter rate (default value is 5%).

To find the optimal sp^*, we use *Bayesian Optimization* [29] to tune the sp. For each configuration of sp, the $\mathcal{L}(T, S)$ value is computed after collecting data from 5,000 sessions. We evaluate the parameter selection in Section 6.3.

Discussion: potential online tuning of sp. The sp parameter values selected above are based on online evaluation across representative device and network conditions, providing robust default settings. However, optimal values for sp may change dynamically during playback due to evolving conditions or shifting user preferences. An online tuning approach could periodically compare recent R2C latency and stutter metrics with the user-indicated latency priority, adjusting sp accordingly to maintain the desired performance balance. Such an adaptive controller has not yet been implemented and represents a promising direction for future work.

5.3 Proactive Keyframe Requester

As illustrated in Section 4.3, if a reference frame is lost during transmission, subsequent frames must wait for it to be retransmitted. This head-of-line blocking can cause excessive latency due to passive waiting in the jitter buffer. For instance, the number of blocked frames can reach up to 29 frames (481 ms), as shown in Figure 13, which can significantly impact latency.

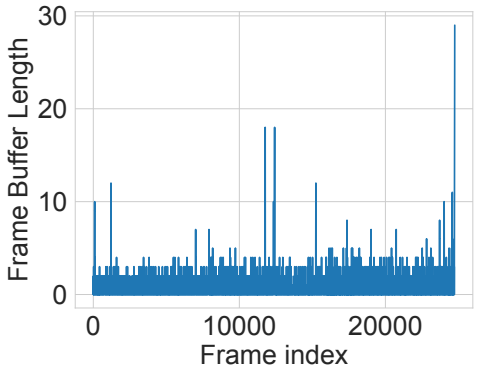


Fig. 13. A case of jitter buffer length dynamics under wireless network.

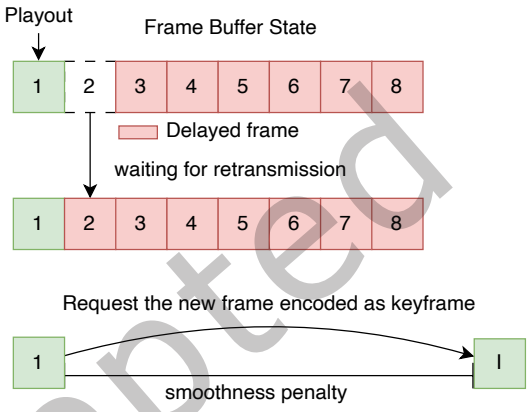


Fig. 14. A comparative illustration of waiting for retransmission and proactively requesting a keyframe.

One possible solution for the client to reduce latency in this case is to *drop certain waiting frames and request a new keyframe*, which can be decoded independently. However, in the default strategy, the client only requests a keyframe when the frame cannot be decoded, usually after a stalling event has already occurred. Therefore, we introduce a proactive keyframe request mechanism.

This mechanism may seem promising to avoid passive waiting. However, it does not necessarily reduce latency since larger keyframes require more time to transmit. Additionally, dropping too many frames will significantly affect smoothness. Figure 14 illustrates the two strategies: waiting for retransmission versus proactively requesting a new keyframe. To this end, we develop *two cost functions*, U_{Wait} and U_{Req}, to determine whether to wait for the retransmission of a lost frame or to proactively request a new keyframe and drop waiting frames.

The first function U_{Wait} measures the latency cost of passive waiting, including the waiting time of the retransmission and the decoding latency of existing frames. The following equation provides the definition :

$$U_{Wait} = T_{RTT} + (Q + 1) \times \bar{T}_{Dec}, \tag{9}$$

where T_{RTT} is the round-trip time (RTT) between the client and the server. Here, we assume that the lost packets can be retransmitted in this RTT. $\bar{T}_{Dec}$ is the average decoding latency of each frame, and Q is the current frame count in the buffer. The term $Q + 1$ accounts for the missing frame when estimating the decoding latency.

The second function U_{Req} measures the cost of proactively requesting a new keyframe. It accounts for the round-trip time (T_{RTT}), the estimated transmission time of the new keyframe ($L_{max}/\hat{C}$), the average decoding latency of the frame ($\bar{T}_{Dec}$), and the smoothness penalty for dropping existing frames ($\lambda \times Q$). This function is defined as:

$$U_{Req} = T_{RTT} + L_{max}/\hat{C} + \bar{T}_{Dec} + \lambda \times Q, \tag{10}$$

Table 3. Symbols and their meanings in Algorithm 1.

Category	Variable	Meaning
Application specific	B_{max}	Maximum buffer level threshold
	λ	Smooth penalty for dropping one frame
	sp	Smooth parameter
Receiver internal state	$\bar{T}_{Dec}$	Average decoding latency
	L_{avg}	Average size of received frames
	L_{max}	Maximum size of received frames
	L_{var}	Variance of frame sizes
	Q	Current frame count in the buffer
	I	Frame interval
	$\hat{C}$	Estimated bandwidth

where all terms are expressed in milliseconds. The smoothness penalty λ represents the latency-equivalent cost of dropping one frame. Since latency degrades QoE more than a brief loss of smoothness [45], the penalty should stay below the 16.7 ms frame interval at 60 fps. We therefore set $\lambda = \frac{1}{3} \times 16.7 \approx 5$(ms).

In practice, when the client receives each video frame, it compares the two cost functions and follows the lower-cost one as the action. Note that T_{RTT} appears in both functions and can be cancelled out, so it does not need to be calculated.

5.4 Putting Everything Together

Algorithm 1 shows the complete algorithm of JitBright, working as follows: *(i)* Retrieving the smooth parameter sp as input from the latency diagnostics. *(ii)* calculating the target level B of the jitter buffer based on Equations 2, 4, and 6, and the maximum buffer level threshold B_{max} (default is 7 frames, i.e., $B_{max} = 7 \times I \approx 116$ ms), corresponding to Lines 1-4 in Algorithm 1; *(iii)* determining whether to passively wait for the retransmission or to proactively request a new keyframe by comparing U_{Wait} (Equation 9) and U_{Req} (Equation 10), corresponding to Lines 5-10 in Algorithm 1. The meaning of the symbols is listed in Table 3.

5.5 Implementation

We implement JitBright by modifying the sender's pixel streaming configuration to change its keyframe frequency. We modify the receiver side WebRTC module, especially *VideoReceiveStream2* [40] and *JitterEstimator* [13] class to support proactive keyframe request and adaptive gain. Additionally, we modified the *PeerConnectionInterface* [41] to obtain the device and network type, which is provided by the application.

Local controlled testbed. To evaluate JitBright in the controlled environment, we set up a local testbed. The client uses a Chromium browser running on a MacBook Pro with an M1 Pro processor, and the server is equipped with an i9-13900KF CPU and an NVidia 3090 GPU. We modify the WebRTC module in the Chromium browser to record the latency of each internal module on the client side. The client connects with the server via WiFi connections. Evaluations are performed in both moving and stationary scenarios. In the moving scenario, the client device moves along a fixed trajectory at walking speed within an office floor with 30 WiFi access points, in which case the bandwidth fluctuates due to handovers between WiFi access points. The trajectory can be traversed in three minutes. In the stationary scenario, the client and the server were placed adjacent to each other. Each test is repeated ten times to eliminate random errors in the environment, and the average results are recorded.

Algorithm 1 Jitter Buffer Optimization Strategy

input: B_{max}; λ; sp; $\bar{T}_{Dec}$; L_{max}; L_{avg}; L_{var}; Q; I; $\hat{C}$
output: B: target buffer level; *enable_req*: if enabling proactively requesting
On inserting frame into the jitter buffer

$S = sp \times I \times \hat{C}$
$gain = L_{var}/S^2$
$B_{test} = gain \times (L_{max} - L_{avg}, 0)/\hat{C}$
$B = \min(B_{max}, B_{test})$
enable_req = False
$U_{Wait} = (Q + 1) \times \bar{T}_{Dec}$
$U_{Req} = L_{max}/\hat{C} + \bar{T}_{Dec} + \lambda \times Q$
if $U_{Wait} > U_{Req}$ **then**
    enable_req= True
end if
return B; *enable_req*

Online system deployment. The built-in browser engine in the application has integrated the webrtc library that implements JitBright. In the online platform (as described in Section 4.1), we executed an A/B test to evaluate the Default strategy against the JitBright strategy using a parameter setting of λ=5 and different parameter sp as defined in Table 5.

6 EVALUATION

This section extensively evaluates JitBright in both the controlled testbed and the wild mobile Internet.

6.1 Setup

Video setting. Our cloud rendering system maintains a constant video resolution of 1560x720 and a frame rate of 60 fps. In this case, the frame interval will remain constant at 16.7ms if played at a consistent speed. The video bitrate is dynamically determined by GCC [6, 30], the default adaptive bitrate mechanism in WebRTC.

Performance metrics. We use two types of metrics to evaluate JitBirght: *(i) MTP and R2C latency*, and *(ii) frame stutter rate*, to measure the probability of a frame arriving later than two encoding frame intervals. This is indicated by the playout (rendering) interval between two consecutive frames exceeding 34 ms.

Baselines. To evaluate the design choice of JitBright, we implement four representative jitter buffer management baselines:

- *Default WebRTC*: The default jitter buffer management strategy in WebRTC-based cloud rendering systems. It includes a periodic (every 300 frames) keyframe insertion from the server. We abbreviate this strategy as *Default*.
- *WebRTC w/o Keyframe*: In this setting, the server does not proactively insert periodic keyframes. Instead, a new keyframe is generated only when the client detects that a frame cannot be decoded and sends a Picture Loss Indication (PLI) message via an RTCP packet, as defined in RFC 4585 [28].
- *0 Buffer*: Disabling the jitter buffer in *Default* by decoding a frame as soon as it is decodable (i.e., its reference frame exists). This strategy is common in previous studies that pursue extremely low latency, while it may compromise smoothness [19, 24].
- *0 Buffer w/ Proactive Keyframe*: *0 Buffer* with proactive keyframe request, which also indicates JitBright disabling the jitter buffer.

6.2 Balancing Latency and Smoothness

We start by evaluating JitBright in the controlled testbed (Section 5.5). Here, we conduct tests in the moving scenario to introduce network dynamics. The results are normalized by the maximum values. As shown in Figure 15, JitBright performs superior over these baselines, striking an optimal balance between R2C latency and playout smoothness. Specifically, it demonstrates an average R2C latency of 50 ms and a stutter rate of 2.8%.

The Default strategy tends to maintain a high buffer level, thus achieving the lowest stutter rate and the highest R2C latency (115 ms). This observation aligns with our analysis in Section 4.4. In contrast, by disabling the jitter buffer, the 0 Buffer and 0 Buffer w/ Proactive Keyframe strategies achieve the lowest R2C latency, however, significantly hindering the smooth playout. Additionally, the WebRTC w/o Keyframe strategy balances the requirements between low latency (67 ms) and smoothness (with 3.6% stutter rate), but it still underperforms JitBirght.

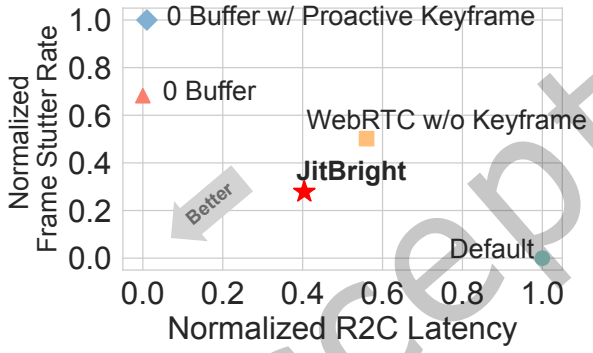


Fig. 15. Performance of JitBright vs. baselines in controlled experiments.

Tail-latency comparison of key-frame strategies. Figure 16 shows that the 90th-percentile buffering latency of JitBright, *Default*, and *WebRTC w/o Keyframe* is 16.0 ms, 105.0 ms, and 70.0 ms, respectively. *Default* inserts a periodic keyframe every 300 frames in addition to any PLI-triggered frames; when the link degrades, these large periodic keyframes take longer to transfer, and the client must passively wait for them, leading to the longest tail. *WebRTC w/o Keyframe* disables periodic keyframes and depends solely on PLI, but the request is sent only after the decoder detects a missing reference, so several dependent frames have already queued up; the extra round-trip before the new keyframe arrives shortens the tail relative to *Default* but still leaves 70 ms at the 90th percentile. JitBright limits active waiting through its adaptive gain and, more importantly, sends a proactive keyframe request before the buffer grows, which confines the P90 tail latency to just one frame interval (16 ms). These results indicate that trimming unnecessary periodic keyframes and triggering an early request, guided by the gain, are both crucial for reducing tail latency.

6.3 Parameter Selection

Next, we evaluate JitBright with different values of the smooth parameter *sp* in Equation 4 to identify the optimal setting. Adjusting *sp* achieves a trade-off between playout smoothness and latency. Specifically, a larger value of *sp* results in lower latency, while a smaller value leads to better smoothness.

Table 4. Performance of different *sp* settings in JitBright.

Strategy	R2C latency	Stutter Rate
JitBright(1)	37.6 ms	3.8%
JitBright(0.5)	38.4 ms	3.7%

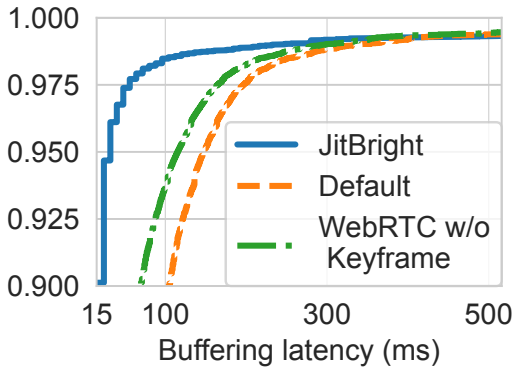


Fig. 16. CDF of buffering latency under three key-frame control strategies.

We choose $sp = 1$ and $sp = 0.5$ for JitBright and test them in the controlled testbed in a stable scenario, denoted as *JitBright(1)* and *JitBright(0.5)*, respectively. The results are presented in Table 4. The two values of sp exhibit similarly in both average R2C latency and stutter rate.

We further validated the optimal selection of the sp parameter across different network and device scenarios. The fine-tuned results are shown in Table 5. For high-end devices, the sp parameter is larger to optimize R2C latency, while for lower-end devices, the sp is smaller. Additionally, the sp for WiFi and 5G networks is higher than for 4G networks to accommodate longer tail latencies, as observed in Section 4.5.

Table 5. The parameter selection of sp for different network and device types

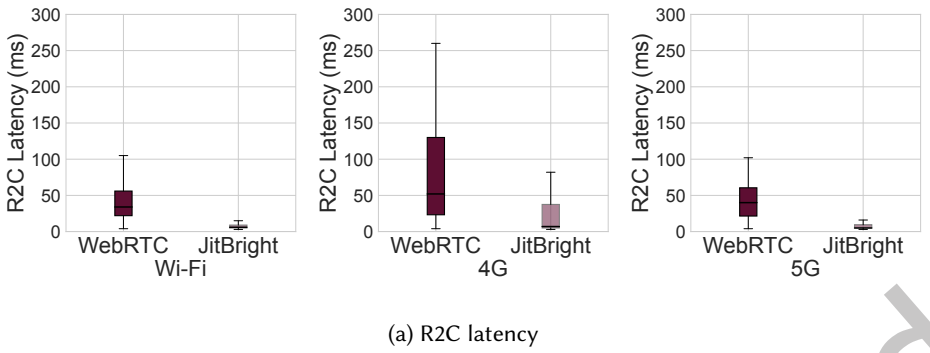
	WiFi	4G	5G
High	1	1	1
Mid	0.8	0.7	0.75
Low	0.6	0.3	0.5

6.4 Large-scale Online A/B Tests

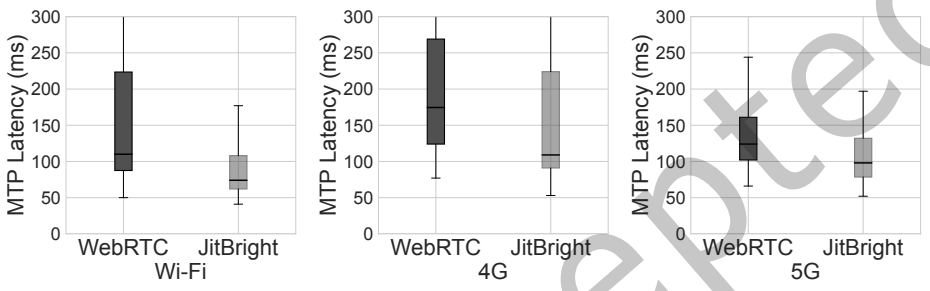
We finally report our large-scale A/B test results of deploying JitBright on the online cloud rendering system. The system randomly assigns JitBright or the Default strategy to each user. Our evaluation was conducted from October 8 to December 15, 2023, with 591,672 sessions. Various types of mobile devices were considered. The evaluation also involved various mobile access networks, including 4G, 5G, and WiFi.

Reducing MTP latency for diverse networks and devices. JitBright reduces R2C latency across all network types, consequently lowering MTP latency, as shown in Figure 17. For WiFi, JitBright reduces the P50 R2C latency by 82.4% and P50 MTP latency by 32.7% compared to WebRTC. On 4G networks,JitBright achieves an 86.5% reduction in P50 R2C latency and a 37.6% reduction in P50 MTP latency. For 5G, JitBright reduces P50 R2C latency by 87.5% and P50 MTP latency by 21.0%. JitBright shows a significant reduction in R2C tail latency (90th percentile) across WiFi (52.6%), 4G (11.3%), and 5G networks (62.4%). The reduction in tail latency is the smallest on 4G networks because the sp is lower on 4G to optimize smoothness, as explained in Section 6.3. Additionally, the longer network tail latency in 4G networks affects the frame arrival process, contributing to higher overall latency, as observed in Section 4.5.

JitBright also reduces R2C latency across all device types, consequently lowering MTP latency, as shown in Figure 18. For high-end devices, JitBright reduces P50 R2C latency by 82.4% and P50 MTP latency by 32.7%



(a) R2C latency



(b) MTP latency

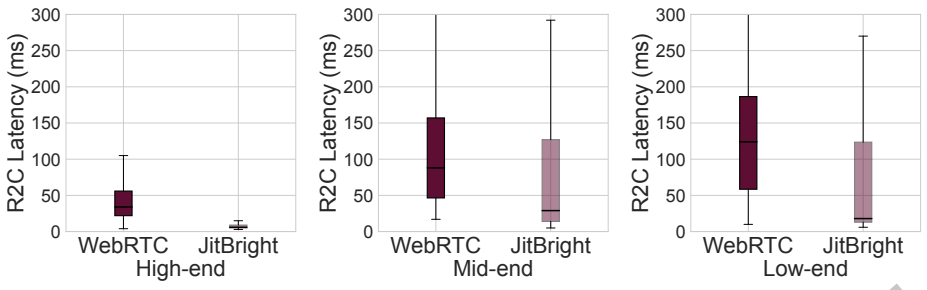
Fig. 17. Comparison of R2C and MTP latencies between WebRTC and JitBright across three different network types. The box represents the interquartile range (IQR). The line inside the box marks the median (50th percentile). The whiskers extend to 1.5 times the IQR from the 25th and 75th percentiles.

compared to WebRTC. For mid-end devices, JitBright achieves a 67.0% reduction in P50 R2C latency and a 25.2% reduction in P50 MTP latency. On low-end devices, JitBright reduces P50 R2C latency by 85.5% and P50 MTP latency by 39.4%. JitBright also demonstrates a reduction in R2C tail latency (90th percentile) across all device levels. For high-end devices, JitBright reduces R2C tail latency by 52.6%. On mid-end devices, the reduction is 22.4%, and for low-end devices, it is 4.0%. The smallest reduction on low-end devices can be attributed to their generally higher baseline latency and more conservative smooth parameter *sp*, as explained in Section 6.3.

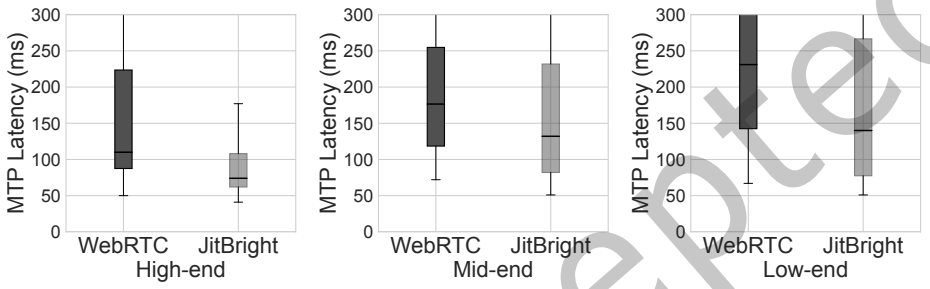
Enabling smooth playout. We also investigate whether JitBright can ensure smoothness. To evaluate the smoothness across all sessions, we choose the session freeze rate as the smoothness metric, defined as the percentage of sessions that experienced at least one freeze event. The freeze event count is originally provided in WebRTC [43]. Our online results show that JitBright effectively reduces the video freeze rate, from 2.4%-2.8% to 0.4%-1.0%.

6.5 Overhead Evaluation

We profiled the two per-frame routines of JitBright: (i) an *adaptive-gain update* that tunes the jitter-buffer size using current network statistics, and (ii) an *enable-request check* that decides whether to send a proactive key-frame request. Both routines, implemented in C++, were instrumented with `std::chrono` and executed 200,000



(a) R2C latency



(b) MTP latency

Fig. 18. Comparison of R2C and MTP latencies between WebRTC and JitBright across three different device grades.

times on two representative devices: an Android smartphone with a Snapdragon 888 (8 cores, 60 fps display) and a desktop PC with an Intel i9-13900KF CPU running Windows 10 at 60 fps.

Results. Table 6 lists the average and worst-case execution time per frame. On the mobile device, the two routines together cost only 0.5 µs on average, while the desktop overhead is even smaller. The implementation maintains only a few scalar variables, resulting in a memory footprint below 8 KB.

The overhead is low because these routines leverage existing metrics (RTT, frame size, decode time) provided by WebRTC, performing only lightweight arithmetic and comparisons, thus incurring negligible CPU overhead on the client.

Table 6. Per-frame CPU cost of JitBright

Platform	adaptive-gain update		enable-request check	
	Avg (μs)	Max (μs)	Avg (μs)	Max (μs)
Snapdragon 888 / 60 fps	0.3	47.1	0.2	38.5
i9-13900KF / 60 fps	0.2	33.8	0.2	45.9

7 RELATED WORK

Measurement in Low-Latency Media Systems. Several studies have focused on measuring latency in cloud gaming systems [7, 12, 19, 25, 45] or live streaming system [49, 50, 53]. Specifically, GamingAnywhere [19] implements a cloud gaming system and conducts latency measurements across various modules at both the sender and receiver end. However, this measurement were performed in controlled experimental environments. Although large-scale measurements exist for live streaming systems [49, 50, 53], the interaction modalities and latency requirements originally differ from those of cloud rendering systems. In contrast, our measurements originate from extensive real-world deployments in online real-time cloud rendering, exclusively involving mobile devices.

Optimization of Low-Latency Media Systems. To address the latency issues caused by client-side jitter buffers, we examine both audio and video jitter buffer techniques in-depth, as detailed in [9, 48] and [34, 52]. However, the approach outlined in voice does not directly apply to the video domain. To the best of our knowledge, the most relevant study to our research is conducted by Zhao et al. [52]. Their methodology primarily focuses on reducing frame jitter buffer latency in live streaming environments. However, it does not address the latency issues due to packet loss.

Client-side latency caused by long decoding latency is discussed in AFR [24]. Whereas AFR concentrates on the latency arising from decoder queuing since it uses a near 0-buffer strategy in its client. Such strategy harms playout smoothness as observed in Section 6.2.

8 CONCLUSION

This paper introduces JitBright, an adaptive jitter buffer optimization strategy aimed at reducing Motion-To-Photon (MTP) latency in mobile cloud rendering. By addressing Receive-To-Composition (R2C) latency, the primary contributor to MTP delay, JitBright optimizes buffer management with adaptive gain control and proactive keyframe requests. Evaluations across over 591,000 sessions showed a significant reduction in R2C latency and an improvement in playback smoothness. These results demonstrate JitBright's effectiveness in balancing low latency and smooth playout, with the potential for further refinements in dynamic network conditions.

REFERENCES

[1] 2007. RFC 4960: Stream Control Transmission Protocol. https://datatracker.ietf.org/doc/html/rfc4960.

[2] 2022. WebRTC Jitter Estimator Implementation. https://source.chromium.org/chromium/chromium/src/+/refs/tags/119.0.6045.169:third_party/webrtc/modules/video_coding/timing/jitter_estimator.cc;l=47.

[3] Alibaba 2024. *Alibaba Cloud Rendering*. Alibaba. https://www.alibabacloud.com/blog/3d-rendering-in-tmall-cutting-edge-tech-to-accelerate-image-preview-30x_596811

[4] Microsoft 2024. *Azure Remote Rendering*. Microsoft. https://azure.microsoft.com/en-us/products/remote-rendering/

[5] Ahmad Alhilal, Tristan Braud, Bo Han, and Pan Hui. 2022. Nebula: Reliable low-latency video transmission for mobile cloud gaming. In *Proceedings of the ACM Web Conference 2022*. 3407–3417.

[6] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo. 2016. Analysis and design of the google congestion control for web real-time communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*. 1–12.

[7] Marc Carrascosa and Boris Bellalta. 2022. Cloud-gaming: Analysis of google stadia traffic. *Computer Communications* 188 (2022), 99–116.

[8] Chromium Authors. 2024. WebRTC Video Coding Timing Module Source Code. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/modules/video_coding/timing/timing.h;drc=4a6bf24b15fdb49a018a12af2025321691f87e1a;l=56. Accessed: 2024-02-03.

[9] Y. Cinar, P. Pocta, D. Chambers, and H. Melvin. 2021. Improved jitter buffer management for WebRTC. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 17, 1 (2021), 1–20.

[10] Mark Claypool and Kajal Claypool. 2006. Latency and player actions in online games. *Commun. ACM* 49, 11 (2006), 40–45.

[11] Xavier Corbillon, Ramon Aparicio-Pardo, Nicolas Kuhn, Géraldine Texier, and Gwendal Simon. 2016. Cross-layer scheduler for video streaming over MPTCP. In *Proceedings of the 7th International Conference on Multimedia Systems*. 1–12.

[12] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. 2021. A network analysis on cloud gaming: Stadia, geforce now and psnow. *Network* 1, 3 (2021), 247–260.
[13] WebRTC Jitter Estimator. 2024. https://webrtc.googlesource.com/src/+/refs/heads/main/modules/video_coding/timing/jitter_estimator.cc. Accessed: 2024-02-03.
[14] Epic Games. 2019. Unreal Engine. https://www.unrealengine.com.
[15] Epic Games. 2023. Pixel Streaming default keyframe interval. https://github.com/EpicGames/UnrealEngine/blob/5.2/Engine/Plugins/Media/PixelStreaming/Source/PixelStreaming/Private/Settings.cpp#L85.
[16] Epic Games. 2024. Unreal Engine PixelStreaming Docs. https://docs.unrealengine.com/5.0/en-US/pixel-streaming-in-unreal-engine/. Accessed: 2024-02-03.
[17] Geekbench. 2024. Geekbench CPU Benchmark. https://browser.geekbench.com/v6/cpu. Accessed: 2024-02-02.
[18] Donghyeok Ho, Hyungnam Kim, Wan Kim, Youngho Park, Kyung-Ah Chang, Hyogun Lee, and Hwangjun Song. 2017. Mobile Cloud-Based Interactive 3D Rendering and Streaming System Over Heterogeneous Wireless Networks. *IEEE Transactions on Circuits and Systems for Video Technology* 27 (2017), 95–109. https://doi.org/10.1109/TCSVT.2016.2565902
[19] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. GamingAnywhere: An open cloud gaming system. In *Proceedings of the 4th ACM multimedia systems conference*. 36–47.
[20] IETF. 2023. Real-Time Communication in WEB-browsers (rtcweb). https://datatracker.ietf.org/wg/rtcweb/about/.
[21] Zenja Ivkovic, Ian Stavness, Carl Gutwin, and Steven Sutcliffe. 2015. Quantifying and mitigating the negative effects of local latencies on aiming in 3d shooter games. In *Proceedings of the 33rd annual acm conference on human factors in computing systems*. 135–144.
[22] Gerui Lv, Qinghua Wu, Weiran Wang, Zhenyu Li, and Gaogang Xie. 2022. Lumos: towards Better Video Streaming QoE through Accurate Throughput Prediction. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*. 650–659. https://doi.org/10.1109/INFOCOM48880.2022.9796948
[23] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. 2022. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/3544216.3544225
[24] Z. Meng, T. Wang, Y. Shen, B. Wang, M. Xu, R. Han, et al. 2023. Enabling High Quality Real-Time Communications with Adaptive Frame-Rate. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1429–1450.
[25] Florian Metzger, Stefan Geißler, Alexej Grigorjew, Frank Loh, Christian Moldovan, Michael Seufert, and Tobias Hoßfeld. 2022. An introduction to online video game qos and qoe influencing factors. *IEEE Communications Surveys & Tutorials* 24, 3 (2022), 1894–1925.
[26] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.
[27] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. 2020. Lumos5G: Mapping and predicting commercial mmWave 5G throughput. In *Proceedings of the ACM internet measurement conference*. 176–193.
[28] Jörg Ott, Stephan Wenger, Carsten Burmeister, José Rey, and Noriyuki Sato. 2006. Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF). RFC 4585. https://doi.org/10.17487/RFC4585
[29] Martin Pelikan and Martin Pelikan. 2005. Bayesian optimization algorithm. *Hierarchical Bayesian optimization algorithm: toward a new generation of evolutionary algorithms* (2005), 31–48.
[30] Chromuim Project. 2024. WebRTC-GCC. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/modules/congestion_controller/goog_cc/.
[31] WebRTC project. 2024. https://webrtc.org/. Accessed: 2024-02-03.
[32] Yanyuan Qin, Shuai Hao, K. R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. 2018. ABR streaming of VBR-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 366–378. https://doi.org/10.1145/3281411.3281439
[33] Devdeep Ray, Vicente Bobadilla Riquelme, and Srinivasan Seshan. 2022. Prism: Handling Packet Loss for Ultra-low Latency Video. In *Proceedings of the 30th ACM International Conference on Multimedia*. 3104–3114.
[34] Luisa Repele, Riccardo Muradore, Davide Quaglia, and Paolo Fiorini. 2013. Improving performance of networked control systems by using adaptive buffering. *IEEE Transactions on Industrial Electronics* 61, 9 (2013), 4847–4856.
[35] Expert Market Research. 2024. Visualisation and 3D Rendering Market. https://www.expertmarketresearch.com/reports/visualisation-and-3d-rendering-market. Accessed: 2024-02-03.
[36] Saeed Shafiee Sabet, Steven Schmidt, Saman Zadtootaghaj, Babak Naderi, Carsten Griwodz, and Sebastian Möller. 2020. 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* (Istanbul, Turkey) *(MMSys '20)*. Association for Computing Machinery, New York, NY, USA, 15–25. https://doi.org/10.1145/3339825.3391855

[37] P. Seeling and M. Reisslein. 2011. Video transport evaluation with H. 264 video traces. *IEEE Communications Surveys & Tutorials* 14, 4 (2011), 1142–1165.
[38] Zhaowei Tan, Jinghao Zhao, Yuanjie Li, Yifei Xu, and Songwu Lu. 2021. {Device-Based}{LTE} latency reduction at the application layer. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 471–486.
[39] Threekit. 2024. How Real-Time Rendering in Ecommerce Enhances Customer Decision Making. https://www.threekit.com/blog/how-real-time-rendering-enhances-customer-decision-making-in-e-commerce. Accessed: 2024-02-01.
[40] WebRTC. 2024. video/video_receive_stream2.cc. https://webrtc.googlesource.com/src/+/refs/heads/main/video/video_receive_stream2.cc. Accessed: 2024-02-03.
[41] WebRTC Project. [n. d.]. PeerConnection Interface. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/api/peer_connection_interface.h. Accessed: 2024-05-21.
[42] Wikipedia contributors. 2024. WebRTC - Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/WebRTC#Support. Accessed: 2024-02-02.
[43] World Wide Web Consortium. 2023. WebRTC Statistics API: Freeze Count Definition. https://www.w3.org/TR/webrtc-stats/#dom-rtcinboundrtpstreamstats-freezecount. Accessed: 2024-02-02.
[44] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, et al. 2021. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference*. 37–53.
[45] Xiaokun Xu and Mark Claypool. 2024. User Study-based Models of Game Player Quality of Experience with Frame Display Time Variation. In *Proceedings of the 15th ACM Multimedia Systems Conference*. 210–220.
[46] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. 2020. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 495–511. https://www.usenix.org/conference/nsdi20/presentation/yan
[47] Encheng Yu, Jianer Zhou, Zhenyu Li, Gareth Tyson, Weichao Li, Xinyi Zhang, Zhiwei Xu, and Gaogang Xie. 2024. Mustang: Improving QoE for Real-Time Video in Cellular Networks by Masking Jitter. *ACM Transactions on Multimedia Computing, Communications and Applications* (2024).
[48] Minglan Yuan. 2019. Jitter buffer control algorithm and simulation based on network traffic prediction. *International Journal of Wireless Information Networks* 26, 3 (2019), 133–142.
[49] H. Zhang, A. Zhou, Y. Hu, C. Li, G. Wang, X. Zhang, et al. 2021. Loki: improving long tail performance of learning-based real-time video adaptation by fusing rule-based models. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*. 775–788.
[50] H. Zhang, A. Zhou, J. Lu, R. Ma, Y. Hu, C. Li, et al. 2020. OnRL: improving mobile video telephony via online reinforcement learning. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–14.
[51] Yuankang Zhao, Qinghua Wu, Gerui Lv, Furong Yang, Jiuhai Zhang, Feng Peng, Yanmei Liu, Zhenyu Li, Ying Chen, Hongyu Guo, and Gaogang Xie. 2024. JitBright: towards Low-Latency Mobile Cloud Rendering through Jitter Buffer Optimization. In *Proceedings of the 34th Edition of the Workshop on Network and Operating System Support for Digital Audio and Video* (Bari, Italy) *(NOSSDAV '24)*. Association for Computing Machinery, New York, NY, USA, 36–42. https://doi.org/10.1145/3651863.3651881
[52] Y. Zhao, A. Zhou, and X. Chen. 2020. Reducing latency in interactive live video chat using dynamic reduction factor. In *2020 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1–6.
[53] A. Zhou, H. Zhang, G. Su, L. Wu, R. Ma, Z. Meng, et al. 2019. Learning to coordinate video codec with transport protocol for mobile video telephony. In *The 25th Annual International Conference on Mobile Computing and Networking*. 1–16.