# JitBright: towards Low-Latency Mobile Cloud Rendering through Jitter Buffer Optimization

Yuankang Zhao[†§A], Qinghua Wu[†§‡], Gerui Lv[†§],Furong Yang[†A], Jiuhai Zhang[A], Feng Peng[A],Yanmei Liu[A], Zhenyu Li[†§‡], Ying Chen[A], Hongyu Guo[A], Gaogang Xie[§¶]
[†]Institute of Computing Technology, Chinese Academy of Sciences
[§]University of Chinese Academy of Sciences [‡]Purple Mountain Laboratories
[¶]Computer Network Information Center, Chinese Academy of Sciences
[A]Alibaba Group

## ABSTRACT

Low-latency cloud rendering services use high-performance servers to provide mobile device users with exquisite graphics and convenient access experiences. Due to the complexity of the system and the diversity of impacting factors, identifying system bottlenecks has become a significant challenge. To demystify system performance, we build an online cloud rendering system to measure the latency distribution of its key components.

Our real-world measurement study reveals that the primary factor causing increased motion-to-photon (MTP) latency is the receive-to-composition (R2C) latency at the client, which is primarily caused by the ineffective jitter buffer management strategy. Based on these findings, we propose JitBright, a systematic jitter buffer optimization that deflates MTP latency. JitBright incorporates adaptive gain and proactive keyframe requests. Large-scale A/B tests involving over 12,000 users demonstrate that JitBright successfully reduces MTP latency while improving playout smoothness. Specifically, JitBright increases the proportion of sessions that meet MTP latency requirements by 6%-27%.

## CCS CONCEPTS

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

## KEYWORDS

Mobile Cloud Rendering, Low Latency, Motion-to-photon Latency, Jitter Buffer

**ACM Reference Format:**
Yuankang Zhao[†§A], Qinghua Wu[†§‡], Gerui Lv[†§],Furong Yang[†A], Jiuhai Zhang[A], Feng Peng[A],Yanmei Liu[A], Zhenyu Li[†§‡], Ying Chen[A], Hongyu Guo[A], Gaogang Xie[§¶]. 2024. JitBright: towards Low-Latency Mobile Cloud Rendering through Jitter Buffer Optimization . In *The 34th edition of the Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '24), April 15–18, 2024, Bari, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3651863.3651881

Corresponding author: Gaogang Xie.


*NOSSDAV '24, April 15–18, 2024, Bari, Italy*

ACM ISBN 979-8-4007-0613-4/24/04
https://doi.org/10.1145/3651863.3651881

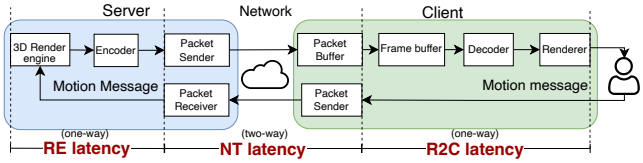


**Figure 1: Cloud rendering architecture and latency distribution.**

## 1 INTRODUCTION

Real-time 3D rendering is significantly impacting e-commerce businesses by enhancing the customer shopping experience and offering new ways to engage with products [33]. Cloud rendering technology renders and streams video frames on servers to the mobile client. This enables users to access 3D visuals in real-time through mobile devices with limited computing resources. As a result, cloud rendering is experiencing a remarkable growth in popularity [2, 3].

Cloud rendering demands stringent latency requirements due to the necessity for user interaction. The demand for interaction requires 100 ms [20] to 150 ms [9] Motion-To-Photon (MTP) latency, which is the delay between user input (or motion) and the resulting image update (or photon) on a display [4]. 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 top e-commerce APP in China (Section 2), we have identified the R2C delay as the primary factor influencing MTP latency. R2C delay is produced in three processes: buffering, decoding, and rendering. Our deeper analysis indicates that the buffering latency constitutes the major component in most (71.5%) cases. We further outline the root causes of buffering latency into *active waiting* and *passive waiting* in the frame buffer (i.e., jitter buffer). Active waiting occurs when a frame is ready for decoding but waits to ensure smooth playout. Passive waiting happens when a frame must wait for the arrival of a reference frame it relies on for decoding.

In this paper, we introduce JitBright, a systematic jitter buffer optimization strategy to reduce MTP latency by minimizing both active and passive waiting latencies. 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 [27] 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 [18, 23]. However, our evaluations in Section 4.2 indicate that this zore-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 3.1).

*(ii) 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 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 3.2).

JitBright is designed to be lightweight and practical and thus is easy to deploy. We have implemented and developed JitBright in our real-world cloud rendering system (Section 3.4). Large-scale online A/B tests with over 12,000 users demonstrate that JitBright successfully reduces MTP latency while improving playout smoothness (Section 4). Specifically, JitBright increases the proportion of sessions meeting MTP latency requirements (i.e., <150 ms) by 15%-23%, 9%-20%, and 6%-27%, in WiFi, 5G, and 4G networks, respectively. Additionally, JitBright reduces the video freeze rate from 2.4%-2.8% to 0.4%-1.0%.

## 2 MOBILE CLOUD RENDERING PERFORMANCE IN THE WILD

This section presents a comprehensive measurement study on mobile cloud rendering performance, based on a dataset of over 2000 users from a top e-commerce APP in China. Through analysis, we identify the bottleneck of optimizing motion-to-photon (MTP) latency, which motivates the design of JitBright.

### 2.1 Measurement Setup

**Background.** Our measurements are performed on a cloud rendering service deployed inside the Taobao mobile app, a top e-commerce APP in China. Figure 2 depicts the architecture of our cloud rendering system. 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, etc.). 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 [17]. Consequently, the user experience is impacted by MTP latency [31].

**Online cloud rendering system.** The cloud rendering service is built using Unreal Engine [13], a state-of-the-art 3D rendering engine famous for creating immersive 3D visualized environments. The cloud rendering service operates on multiple rendering servers in 4 geographically distributed data centers in China. Each server creates a series of real-time images of 3D models, which are captured into a live video stream using Pixel Streaming [15]. This stream is then composed and transmitted using WebRTC [27]. A separate WebRTC signal server is used to set up WebRTC connections with the client. We choose WebRTC because it already serves as a standard interactive video streaming framework [19] and is widely supported by major web browsers and platforms [35], thus facilitating large-scale deployments.

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.

**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 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 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.

### 2.2 Vivisecting MTP Latency in Mobile Cloud Rendering

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 3 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 [21, 22, 37]. *(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 them, 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 4, respectively. The results show that when MTP latency exceeds the latency requirements

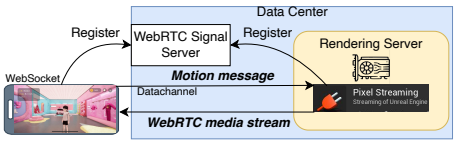


**Figure 2: Online mobile cloud rendering system overview.**

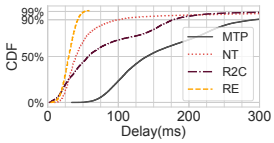


**Figure 3: MTP latency and its three components.**

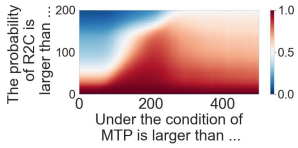


**Figure 4: Conditional probability of R2C and MTP latency.**

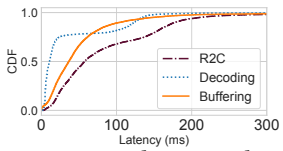


**Figure 5: R2C latency and its two components.**

(i.e., over 150 ms), over 50% of cases experience R2C latency over 100 ms (i.e., 2/3 of 150 ms), leading to the following observation:

***Observation*: Unsatisfactory MTP latency is dominated by inflated R2C latency.**

## 2.3 Key Factors Inflating R2C Latency

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?*

**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 received frames 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 5 shows the latency distribution of each process. Note that rendering latency is omitted because it is always less than 10ms [7]. The results show that **buffering latency is the key factor impacting R2C latency in most (71.5%) cases.**

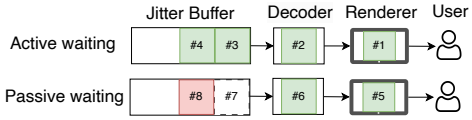


**Figure 6: Active and passive waiting in the jitter buffer.**

**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 6. 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 [10], and is usually caused by packet loss in the transmission [29]. To explain, two types of frames exist in video streaming: 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.

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.**

## 2.4 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 3.4), 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 (not presented due to limited space). 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* [12]. Jitter buffer aims to enable the smooth playout of video frames. Specifically, when the bandwidth suddenly drops or a frame is too large, causing the transmission time to exceed the inter-frame interval (e.g., 16.7ms at 60 fps), 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 [41]), respectively. $\hat{C}$ is the estimated bandwidth, using Kalman Filter [5].

**Performance issue.** In video streaming, the frame size is determined by both the frame type (keyframe or delta frame) and the content complexity [28] when the target encoding bitrate is constant. In particular, a keyframe is generally 4-10 times larger than a delta frame [32], 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 [14, 18]. Under this circumstance, the default strategy is too conservative, resulting in unnecessary active waiting and ultimately inflated R2C latency.

Figure 7 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).

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.

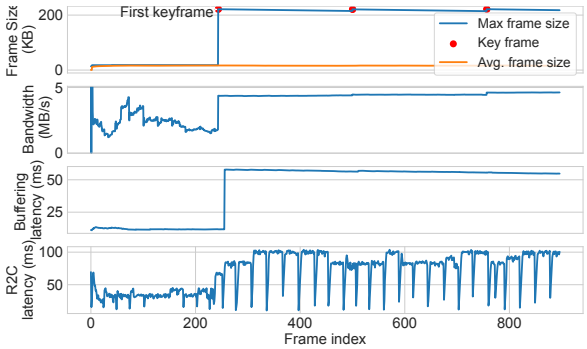


**Figure 7: The impact of maximum frame size and its frequency on the estimated value of jitter delay.**

## 3 DEFLATING MTP LATENCY BY JITBRIGHT

As illustrated in Section 2, the key to reducing MTP latency is to improve the jitter buffer scheduling strategy. The ideal strategy aims to optimize user experience by balancing two conflicting goals: *(i) reducing latency*, to avoid MTP latency inflated than expected (e.g., over 150 ms); and *(ii) ensuring smoothness*, to avoid frames played at inconsistent speed or even stalling.

To this end, we propose JitBright, a systematic jitter buffer optimization strategy that deflates MTP latency while ensuring smooth playout. Specifically, JitBright introduces an adaptive gain and a proactive keyframe request mechanism. The adaptive gain controls the jitter buffer level based on the probability that a frame will not play smoothly, thereby avoiding unnecessary active waiting. On the other hand, the proactive keyframe request removes the decoding dependency between frames, thus reducing passive waiting latency.

### 3.1 Adaptive gain

To balance the requirements of low latency for active waiting and smooth playout, JitBright introduces an adaptive *gain* into the default strategy. That is, transform Equation 1 to the following:

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

**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 $I$). 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 4), $I$ 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 to 1.0 as discussed in Section 4.3.

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

Based on *Chebyshev's inequality* [25], 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.

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

**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 small, 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 4.

There is a special case for this algorithm: when the client requests a keyframe (see Section 3.2), 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.

### 3.2 Proactive Keyframe Request

As illustrated in Section 2.3, 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.

One possible solution in this case, is to *request a new keyframe and drop all the waiting frames in the buffer*. In this way, the latency can be reduced by removing the decoding dependency as a keyframe is 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, JitBright introduces a proactive keyframe request mechanism.

Although requesting keyframes seems promising to avoid passive waiting, it does not necessarily reduce latency because larger keyframes require more time to transmit. Additionally, dropping too many frames will significantly affect smoothness. 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 :

$$U_{Wait} = T_{RTT} + Q \times \bar{T}_{Dec}, \tag{7}$$

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 second function $U_{Req}$ measures the cost of proactively requesting, including the waiting time for a new keyframe to arrive, the decoding latency of this frame, and the cost of dropping existing

**Table 1: Symbols and their meanings in Algorithm 1.**

| Category | Variable | Meaning |
|---|---|---|
| Application specific | $B_{max}$ | Maximum buffer level threshold |
| | $\lambda$ | 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 |

frames. This function is defined as:

$$U_{Req} = T_{RTT} + L_{max}/\hat{C} + \bar{T}_{Dec} + \lambda \times Q, \tag{8}$$

where $L_{max}/\hat{C}$ indicates the estimated transmission time of the new keyframe and $\lambda$ is a smooth penalty of dropping one frame (default value is 5). $T_{RTT}$ indicates the time from when the client sends the request to when it receives the first byte of the new keyframe[1].

In practice, when the client receives each video frame, it compares the two cost functions and follows the lower-cost one as the action[2].

## 3.3 Putting Everything Together

Algorithm 1 shows the complete algorithm of JitBright, working as follows: *(i)* 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; *(ii)* determining whether to passively wait for the retransmission or to proactively request a new keyframe by comparing $U_{Wait}$ (Equation 7) and $U_{Req}$ (Equation 8), corresponding to Lines 5-10 in Algorithm 1. The meaning of the symbols is listed in Table 1.

**Algorithm 1** JitBright: Jitter Buffer Optimization Strategy

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

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

## 3.4 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* [34] and *JitterEstimator* [12] class. The default parameter setting in JitBright is $sp$=1 and $\lambda$=5.

**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 i7-13900K 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. Traversing the trajectory takes 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, and the average results are recorded.

**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 2.1), we executed an A/B test to evaluate the Default strategy against the JitBright.

[1] $T_{RTT}$ in both functions can be canceled out, so it does not need to be calculated.
[2] When requesting a keyframe, if lost packets of previous frames are received, there is an additional cost due to the extra size of an I-frame over a P-frame. However, this item is not included because the probability of such an event is difficult to determine.

# 4 EVALUATION

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

## 4.1 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 [5, 26], 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*: The default jitter buffer management strategy in WebRTC-based cloud rendering systems. It includes a periodic (every 300 frames) keyframe insertion from the server.
- *Default w/o Keyframe*: *Default* without server sending periodic keyframe. In this case, the jitter buffer (as well as the latency) is expected to remain at a low level.
- *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 [18, 23].
- *0 Buffer w/ Proactive Keyframe*: *0 Buffer* with proactive keyframe request, which also indicates JitBright disabling the jitter buffer.

## 4.2 Balancing Latency and Smoothness

We start by evaluating JitBright in the controlled testbed (Section 3.4). Here, we conduct tests in the moving scenario with the dynamic network. The results are normalized by the maximum values.

As shown in Figure 8, JitBright performs superior over these baselines, striking an optimal balance between R2C latency and

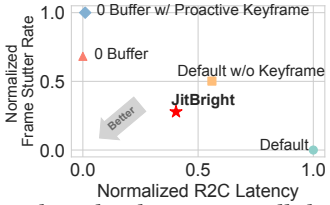


**Figure 8: JitBright vs. baselines in controlled experiments.**

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 2.4. 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 Default w/o Keyframe strategy balances the requirements between low latency (67 ms) and smoothness (with 3.6% stutter rate), but it still underperformers JitBirght.

## 4.3 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 2: 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% |

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 2. The two values of $sp$ exhibit similarly in both average R2C latency and stutter rate. We finally choose $sp = 1$ for its lower latency.

## 4.4 Large-scale Online A/B Test

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 August 3 to September 17, 2023, with over 12,000 volunteer users and a total of 314 hours of video time. Various types of mobile devices were considered. The device grades were categorized into high, medium, and low according to their CPU performance and benchmark scores [16]. Additionally, the evaluation also involved various mobile access networks, including 4G, 5G, and WiFi.

**Reducing MTP latency.** Table 3 presents the performance improvement by JitBright versus the Default strategy across various device grades and mobile access networks. Here, 150 ms is taken as the bound of MTP latency. The online results demonstrate that JitBright improves the proportion of MTP latency below 150 ms ranging from 6% to 27%, successfully deflating the overall latency.

**Enabling smooth playout.** We also investigate whether JitBright can ensure smoothness. Since the stutter rate is difficult to access directly in the online system, we choose another metric, the freeze rate, which is the percentage of sessions that experienced at least one freeze event. The freeze event count is originally provided in WebRTC [36]. Our online results show that JitBright effectively reduces the video freeze rate, from 2.4%-2.8% to 0.4%-1.0%. The figure is omitted due to limited space.

**Table 3: The increase in the percentage of samples with MTP latency <150 ms in the large-scale online A/B test.**

| | WiFi | 4G | 5G |
|---|---|---|---|
| High | +15% | +27% | +9% |
| Mid | +13% | +16% | +20% |
| Low | +23% | +6% | +15% |

# 5 RELATED WORK

**Measurement in Low-Latency Media Systems.** Several studies have focused on measuring latency in cloud gaming systems [6, 11, 18, 24] or live streaming system [39, 40, 42]. Specifically, GamingAnywhere [18] 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 [39, 40, 42], 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 [8, 38] and [30, 41]. 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. [41]. 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 [23]. 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 4.2.

# 6 CONCLUSION

The interactive 3D cloud rendering system requires low Motion-To-Photion (MTP) latency to improve the user experience. Through large-scale online measurements, we find that MTP latency is demonstrated by Receive-To-Composition (R2C) latency, which is fundamentally caused by the ineffective jitter buffer scheduling strategy on the client side. This paper proposes JitBright as a systematic jitter buffer optimization strategy. Real-world A/B tests with over 12,000 users demonstrate that JitBright effectively reduces MTP latency while ensuring smooth playout.

# ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by the National Key R&D Program of China (2022YFB2901800), Natural Science Foundation of China (U20A20180, 62072437 and 62321166652) and Beijing NSF (JQ20024).

# REFERENCES


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

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

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] WebRTC Jitter Estimator. 2024. https://webrtc.googlesource.com/src/+/refs/heads/main/modules/video_coding/timing/jitter_estimator.cc. Accessed: 2024-02-03.

[13] Epic Games. 2019. Unreal Engine. https://www.unrealengine.com.

[14] 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.

[15] Epic Games. 2024. Unreal Engine PixelStreaming Docs. https://docs.unrealengine.com/5.0/en-US/pixel-streaming-in-unreal-engine/. Accessed: 2024-02-03.

[16] Geekbench. 2024. Geekbench CPU Benchmark. https://browser.geekbench.com/v6/cpu. Accessed: 2024-02-02.

[17] 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

[18] 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.

[19] IETF. 2023. Real-Time Communication in WEB-browsers (rtcweb). https://datatracker.ietf.org/wg/rtcweb/about/.

[20] 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.

[21] 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

[22] 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

[23] 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.

[24] 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.

[25] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.

[26] Chromuim Project. 2024. WebRTC-GCC. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/modules/congestion_controller/goog_cc/.

[27] WebRTC project. 2024. https://webrtc.org/. Accessed: 2024-02-03.

[28] 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

[29] 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.

[30] 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.

[31] 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

[32] P. Seeling and M. Reisslein. 2011. Video transport evaluation with H. 264 video traces. *IEEE Communications Surveys & Tutorials* 14, 4 (2011), 1142–1165.

[33] 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.

[34] WebRTC. 2024. video/video_receive_stream2.cc. https://webrtc.googlesource.com/src/+/refs/heads/main/video/video_receive_stream2.cc. Accessed: 2024-02-03.

[35] Wikipedia contributors. 2024. WebRTC - Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/WebRTC#Support. Accessed: 2024-02-02.

[36] 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.

[37] 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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.