ACE: Sending Burstiness Control for High-Quality Real-time Communication

XIANGJIE HUANG, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

JIAYANG XU, Hong Kong University of Science and Technology, Hong Kong, Hong Kong

HAIPING WANG, ByteDance Ltd., Beijing, China

HEBIN YU, ByteDance Ltd., Beijing, China

SANDESH DHAWASKAR SATHYANARAYANA, ByteDance Ltd., Beijing, China

SHU SHI, ByteDance Ltd., Beijing, China

View all

Open Access Support provided by:

Hong Kong University of Science and Technology

ByteDance Ltd.

PDF Download
3718958.3750520.pdf
12 January 2026
Total Citations: 0
Total Downloads: 2060

Published: 08 September 2025

Citation in BibTeX format

SIGCOMM '25: ACM SIGCOMM 2025 Conference
September 8 - 11, 2025
Coimbra, Portugal

Conference Sponsors:
SIGCOMM

ACE: Sending Burstiness Control for High-Quality Real-time Communication

Xiangjie Huang[1,2], Jiayang Xu[1], Haiping Wang[2], Hebin Yu[2]
Sandesh Dhawaskar Sathyanarayana[2], Shu Shi[2], Zili Meng[1]
[1]Hong Kong Univeristy of Science and Technology [2]ByteDance
xhuangdd@connect.ust.hk, jxudn@connect.ust.hk, wanghaiping.paloma@bytedance.com, yuhebin.824@bytedance.com, sandesh.dhawaskar@bytedance.com, shishu.1513@bytedance.com, zilim@ust.hk

Abstract

Modern real-time communication (RTC) demands both ultra-low latency and consistently high visual quality. Yet, as content becomes more dynamic and RTTs shrink, we reveal a previously overlooked problem: long-tail queuing latency in the sender's pacing queue between encoder and network. This phenomenon is rooted in a mismatch between the bursty frame stream produced by the encoder and the smooth traffic expected by the network. Existing approaches trying to smoothen the bitrate inevitably force an undesirable trade-off between latency and video quality. To address this, we propose a dual-control approach that manages both the encoding and transmission burstiness. At the sender, we dynamically adjust the bucket size of a token-based pacer to control burstiness at the granularity of frame level. Within the encoder, we introduce an adaptive complexity mechanism that smoothens frame sizes without sacrificing quality. Trace-driven emulation and real-world experiments show our solution ACE reduces end-to-end 95th percentile latency by up to 43% while maintaining superior visual quality versus the state of the art.

CCS Concepts

• **Networks** → **Cross-layer protocols**; • **Information systems** → **Multimedia streaming**.

Keywords

Real-time communications, pacing, adaptive complexity encoding

ACM Reference Format:
Xiangjie Huang, Jiayang Xu, Haiping Wang, Hebin Yu, Sandesh Dhawaskar Sathyanarayana, Shu Shi, Zili Meng . 2025. ACE: Sending Burstiness Control for High-Quality Real-time Communication. In *ACM SIGCOMM 2025 Conference (SIGCOMM '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3718958.3750520

1 Introduction

Real-time communication (RTC) systems have become indispensable in modern interactive applications, from video conferencing and cloud gaming to remote operations and virtual reality. Advancements in network infrastructure (e.g., edge servers, 5G) have reduced network round-trip times (RTTs) to sub-30 ms [36, 59, 63]. Meanwhile, there are numerous efforts trying to reduce the end-to-end latency for RTC via video codec [18, 37, 43], Congestion Control Algorithms (CCAs) [17, 45, 53, 57], loss recovery [5, 12, 46], and even network layer and link layer optimizations [34, 35, 63]. Unfortunately, users still suffer from latency-related stalls today [5, 42, 63].

One critical yet understudied bottleneck in modern RTC systems is the **pacing latency** between the video encoder and the network. Video streams are generated by encoders at each frame interval, which inherently results in bursty traffic patterns. This burstiness can become particularly pronounced in the context of high-quality RTC applications. For instance, a cloud gaming application operating at a bitrate of 30 Mbps [32, 37] and 30 fps will see each frame comprising over 100 packets. Transmitting such bursts directly into the network elevates the risk of overshooting network buffers. To tame them, WebRTC and similar stacks apply *pacing*, which flattens each frame into a uniform packet stream at a steady rate [6, 16, 51]. While this prevents overshooting network buffers, it introduces additional pacing latency. Ideally, the added pacing latency should not exceed one frame interval(e.g., 33 ms for 30 fps), as the pacing rate is at least the bitrate determined by Congestion Control. However, this is not always true, and we discuss the emergence of pacing latency in §3.1.

In today's context, the issue of pacing latency has grown increasingly critical. First, network RTTs have already shrunk to below the frame interval, making the pacing latency a significant portion of the end-to-end delay. Especially when the bitrate fluctuates, the existing packets in the pacing buffer can introduce extra pacing latency of multiple times of the network RTT. Second, the growing variability in content leads to significant fluctuations in frame sizes, amplifying the impact of pacing delays. While CCAs set a target bitrate for video encoders, generated frame sizes fluctuate around this target due to content variability. As a consequence, from measurements of our nationwide cloud gaming service involving over O(100K) sessions, pacing latency has emerged as a key contributor to video stalls, as we will discuss in §3.1. Meanwhile, another important observation is that much of this pacing latency is unnecessary: the CCAs employed in low-latency RTC applications typically es-

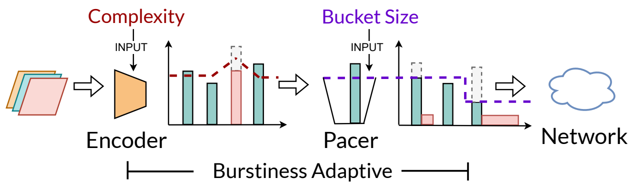


Figure 1: ACE Overview. Comlexity adaptive encoding (ACE-C) and burstiness adaptive pacing (ACE-N) enable dual-control of sending patterns.

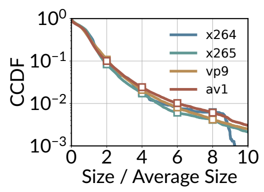


Figure 2: CCDF of encoded frame size

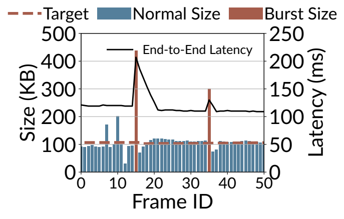


Figure 3: Latency impact by Oversized frames

timate bandwidth conservatively [23, 31]. Due to the conservation bandwidth underestimation of latency-sensitive CCAs, *transiently* allowing some bursts to the network *to a limited extent* might be beneficial to the end-to-end latency.

Unfortunately, existing solutions failed to mitigate the increase in the pacing latency. Notably, the issue is caused by the transient mismatch between the burstiness of the encoding of video frames and the smoothness of pacing packets to the network. Low-latency CCAs [45, 53] can reduce the network RTT by determining how many packets to send in one RTT, but do not optimize how to send these packets at each timepoint. Different bucket-based pacers [19, 33] have heterogeneous features, while the parameters of pacing are still fixed, oscillating between network overflows and high pacing latency. Existing co-design [17, 56] efforts between the codecs and the network reduce the pacing latency by enabling accurate bitrate control, but at the cost of sacrificing the video quality.

This motivates us to address pacing latency by tackling the problem at a finer-grained timescale. We analyze from both the dequeue and enqueue perspectives of the pacing buffer. From the dequeue perspective (packets sent to the network), the sending patterns should be adaptively regulated to achieve two objectives: (i) minimize the risk of network buffer overflow caused by burstiness and (ii) reduce unnecessary pacing latency. This necessitates adapting to the state of the network buffer and determining how much data can be instantly transmitted into the network. From the enqueue perspective (frames generated from the encoder), the encoder must generate video frames as smoothly as possible to avoid oversized frames, which mitigates frame-level burstiness. And meanwhile it should maintain the visual quality required by the application.

To this end, we propose ACE, a burstiness control framework for high-quality RTC that integrates complexity-adaptive encoding (ACE-C) and burstiness-adaptive pacing (ACE-N). The design of ACE is illustrated in Fig. 1. By leveraging this dual-control mechanism, ACE answers two questions:

(1) ***When to burst?*** To reduce overall latency by eliminating pacing delays, we must carefully manage burst transmission. Though bursting can improve latency, it risks exacerbating end-to-end delays if the network buffer overflows, which can lead to massive retransmissions (detailed discussed in §3.3). Preventing such losses is critical – we only want to allow bursts to a limited extent of not to overflow the network. To address this, ACE-N adjusts the bucket size of a token bucket pacer to be adaptive to network conditions. The key factor influencing this decision is the available buffer size in the network, which is highly unpredictable. ACE-N observes the state of the in-network buffer by estimating network buffer occupancy using fine-grained packet arrival patterns. The design of ACE-N is detailed in §4.1.

(2) ***How to compress?*** Existing solutions mitigate the oversized frames by compromising visual quality [17, 56], which is not acceptable for high-quality RTC applications. ACE-C avoids by exploiting the **complexity-size tradeoff** inherent in modern video codecs. By adaptively increasing encoding complexity, ACE-C achieves smaller frame sizes without quality degradation – albeit at the cost of higher encoding time and computational overhead. This approach comes with two challenges: (i) maintaining the practicality by strictly bounding overhead increases, and (ii) predicting whether the increase of encoding time outweighs the decrease of pacing latency. To address these challenges, we first predict frame size before the encoding process by the differences between two frames. Based on this prediction, ACE-C selects only the oversized frames (less than 5%) and adjust appropriate complexity levels to reduce the sum of encoding time and pacing latency. The design details are discussed in §4.2.

We implement ACE on top of WebRTC and a widely-used libx264 encoder and evaluate its performance through both trace-driven emulations and real-world experiments in §6. ACE achieves a 43% reduction in 95th-percentile latency while maintaining VMAF scores [44] equivalent to those of the highest-quality baselines. Our comprehensive evaluation also includes interactions with different CCAs, ablation studies, fairness, and system overheads. Additionally, we implement part of our design, ACE-N, on the RTC engine employed by our cloud gaming application and evaluate its performance through real-world traces. Results show that ACE-N reduces latency by 15% compared to production baselines and maintains the best stall rate and frame rate. The implementation will be open-sourced and can be easily applied as a patch to both WebRTC and libx264 source code.

2 Background

We explain two key phenomena in real-time video encoding: frame size variability and encoding complexity tradeoffs. These form the foundation for ACE's design.

Frame size fluctuation. Modern video codecs inherently produce variable frame sizes even under constant bitrate targets. This variability stems from two compression mechanisms. First, content-dependent redundancy: frames with higher visual complexity, such as those containing intricate textures, require more bits to maintain quality compared to simpler frames like static backgrounds. Second, inter-frame dependencies: predictive coding (e.g., P-frames) creates variable compression ratios depending on reference frame similarity. In this context, only the average bitrate aligns with the target bitrate over time in a long timescale [20], while individual frame sizes can significantly fluctuate around the target bitrate.

We empirically support the observation by experiments. We transcode all the videos from the UGC dataset from YouTube [55] with low latency presets by 4 commonly used video codecs(H264, H265, VP9, AV1) and measure the frame size. As depicted in Fig. 2, all codecs exhibit heavy-tailed distributions - 10% of frames exceed 2× average size, and 1% even exceeding 5× the average size. Importantly, this variability in frame size is not attributable to the distinction between key frames and non-key frames, as RTC rarely has key frames nowadays [40].

Encoding complexity. Video codecs expose complexity parameters that govern computational effort during encoding. These parameters control motion estimation depth (e.g., search range, subpixel precision), mode decision granularity (partition sizes, reference frame

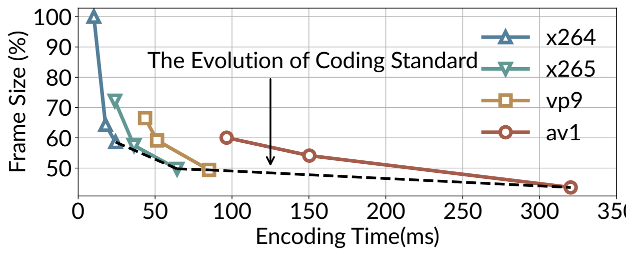


Figure 4: Impact of encoding complexity on frame size and encoding time. The frame size is normalized by the largest one.

selection), and quantization optimizations. Higher complexity settings enable better compression efficiency—smaller frame sizes at equivalent quality—but increase encoding time.

We quantify the relationship between frame size and encoding time across four prominent codecs under the same quality. As illustrated in Fig. 4, all codecs exhibit a trade-off between frame size and encoding time. For example, compared to the lowest complexity, using the highest complexity can reduce frame size by 38%–51%. While advancements in coding standards continue to improve compression efficiency (reducing the minimum achievable frame size, as shown by the dashed line), they do not eliminate this trade-off.

A critical advantage of complexity adaptation lies in its asymmetric impact. Unlike encoding, decoding remains unaffected by complexity choices. As shown in Fig. 5, encoding time escalates from 6 ms to 12 ms with increased complexity, whereas decoding time exhibits minimal variation. Our evaluation also explores the CPU and memory overheads associated with both encoding and decoding (details in the §6 and Appendix. B). The findings indicate that the encoder's CPU usage and memory increase more significantly with higher complexity compared to the decoder's, which remain almost unaffected. This asymmetry enables ACE to optimize encoding decisions without burdening resource-constrained receivers—a crucial property for mobile clients in cloud gaming and VR applications.

3 Motivation and Challenges

In this section, we discuss the motivation to address the problem of pacing latency in §3.1, analyze the limitations of existing solutions in §3.2, and outline the design challenges in both the encoding and sending processes in §3.3.

3.1 Motivation: Controlling Pacing Latency.

Pacing latency has emerged as a significant challenge in real-time communication, particularly in latency-sensitive applications like cloud gaming. Measurements from our production cloud gaming service emphasize the critical need to address this issue. To quantify the impact of pacing latency, we collected and analyzed end-to-end latency breakdowns and video stall events from online-reported data over a full 24-hour period on January 1st, 2025. This dataset encompasses approximately 300,000 sessions, with metrics reported at 2-second intervals. Our analysis specifically focuses on scenarios where the pacer is active. We examined three key latency components: encoding/decoding latency, pacing latency, and transmission latency (measured by RTT). If pacing latency were not a factor contributing to video stalls, its value would remain consistent regardless of stall events, similar to coding latency. However, as shown in Fig. 6, pacing latency during stall events is 60% higher than without stalls

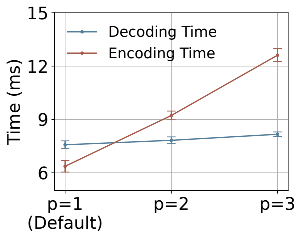


Figure 5: Encoding and decoding Time vs. complexity

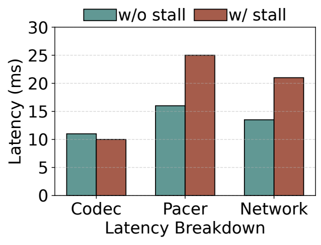


Figure 6: Latency breakdown

and is larger than the network delay (RTT). This observation underscores the strong correlation between pacing latency and video stalls. Notably, even though the latency is averaged over the 2-second reporting interval, the increase during stall events remains evident.

The primary reason for pacing latency arising is **encoder overshoots target rate**. Ideally, pacing latency is at the level of the frame intervals – when the pacer smooths an average-sized frame into the network at the estimated rate, the pacing latency for a frame corresponds to one frame interval (e.g., 33 ms at 30 fps). However, the situation deteriorates when individual frames are oversized. As we discussed in §2, the frame size will still fluctuate to up to 5× for 1% of frames even if the target bitrate is fixed. When frames are transmitted at the pacing rate, the time required to send a frame becomes directly proportional to its size. We provide an example to show how the oversized frame affects the latency, as illustrated in Fig. 3. Despite the average frame size being close to the target (as indicated by the dashed line), the transmission of an oversized frame (highlighted in red) results in a substantial increase in pacing latency. Consequently, the end-to-end latency (solid line) experiences a corresponding surge.

Trend: Pacing latency becomes increasingly important as RTT decreases and frame size variations grow. With the advent of technologies like CDNs, 5G, and Wi-Fi 6, combined with the evolution of congestion control algorithms (CCAs) from traditional approaches (e.g., Cubic/BBR) to modern ones (e.g., GCC/Copa), network conditions have dramatically improved. In the past, RTTs for interactive video streaming often exceeded 100 ms, making pacing latency negligible in comparison. However, today, as RTTs shrink significantly, pacing latency has become a critical factor. Currently, the RTT for interactive video streaming is within 30 ms. For instance, we measure the median RTT in our production cloud gaming service nationwide for 12 months in 2024, covering O(100) games, resulting in O(300M) sessions. The median network RTT of the measurement is only 29.0 ms across all sessions. If we only focus on the sessions with clients and servers in the same region, the median network RTT even drops to 19.6 ms. In this context, at 30 FPS, a frame that is double the average size introduces pacing latency of 67 ms (2×33.3 ms), making it 6× the one-way latency.

We further validated this trend through emulated evaluations on WebRTC, with experimental settings detailed in §6.1. Fig. 7(a) breaks down latency components across RTTs ranging from 160 ms to 10 ms, focusing only on long-tail frames with end-to-end latency exceeding 200 ms. As RTT decreases, pacing latency gradually emerges as the dominant contributor to total delay. Fig. 7(b) fixes the RTT at a low value of 20 ms with various overall latency ranges. Pacing latency accounts for over 60% of the total delay when the overall latency

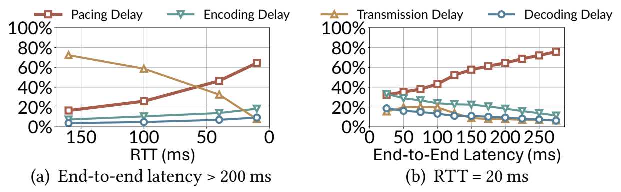


Figure 7: Pacing latency contribution in WebRTC: Dominates at low RTT (a) and high end-to-end latency (b).

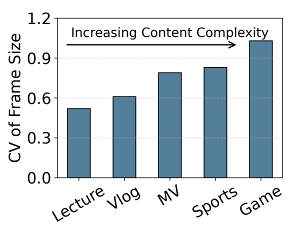


Figure 8: Frame size variation over video contents

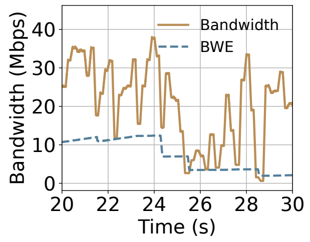


Figure 9: Low-latency CCA always underestimates bandwidth.

reaches 200 ms. In summary, pacing latency becomes significant in scenarios where RTT is low yet overall latency remains high.

Another important trend exacerbating pacing latency is the increasing variability in frame sizes due to the growing complexity of real-time communication content. As shown in Fig. 8, the standard deviation of frame sizes varies across different content types(same settings align with §6) when encoded with the same real-time encoder. We found that as content complexity increases—from *lecture* to *vlog* to *game* videos—the coefficient variation (CV) of frame sizes nearly doubled from 0.56 to 1.03, which intensifies the pacing latency problem.

Issue: Bandwidth underestimation by latency-sensitive CCA makes pacing latency not always necessary. To achieve a consistent low network latency, high-quality RTC applications employ latency-sensitive CCAs such as GCC rather than throughput-oriented CCAs such as Cubic [45, 53]. By being sensitive to subtle network fluctuations, these CCAs prevent queue buildup in the network, but this comes at the cost of reduced network utilization—they consistently underestimate available bandwidth. Such an underestimation is rooted in the fundamental trade-off between queueing delay and utilization in queueing theory [37].

We have seen this observation in our experiments as well. We test GCC over a mixed network condition, including 4G, 5G, and Wi-Fi (details in §6.1) for over 1200 seconds. As shown in Fig. 9, GCC reacts conservatively to network fluctuations, creating a substantial gap between the actual bandwidth and the estimated bandwidth (BWE). In fact, across all traces, GCC is underestimating the available bandwidth in over 90% of the time. This observation aligns with findings from prior studies [23, 31].

Rather than increasing the sending rate to fully utilize bandwidth—a trade-off decision best left to CCA designers—we focus on refining *sub-RTT* sending patterns. While CCAs determine *how many* packets to send within one RTT, they do not optimize *how* these packets are sent within that interval (e.g., via pacing, bursting, or adaptive strategies). This latter question is orthogonal to CCA design.

By optimizing sub-RTT sending patterns, it's possible to avoid excessive pacing latency. Given the bandwidth underestimation inherent to latency-sensitive CCAs, sending packets in short bursts offers two key benefits: (i) it shifts packets from the pacing buffer to router buffers, and (ii) it is unlikely to cause buffer overflow. In such scenarios—where the actual network capacity is typically larger than the CCA's conservative estimate—queueing delays in the network will be shorter than the pacing latency otherwise introduced. In that case, the pacing latency is unnecessary, and sending frames in burst can reduce end-to-end delay. That said, buffer overflow may still occur during network fluctuations, where pacing latency remains necessary. Thus, we are motivated to dynamically adapt sending patterns between pacing and bursting to strike an optimal balance.

3.2 Why Do Existing Solutions Fall Short?

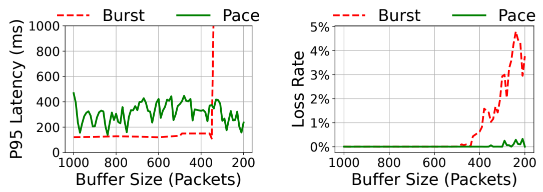
Current methods—such as various pacer designs, refined congestion control, and precise encoding rate control—exhibit fundamental limitations in effectively mitigating the issue of pacing latency.

(1) Limited adaptation to bursting. Various implementations design different pacer structures to schedule data transmission. For instance, WebRTC employs a leaky bucket algorithm to pace data according to the estimated bandwidth, permitting bursts up to 2.5 times the pacing rate for large I-frames. Similarly, some QUIC implementations, such as mvfst [29, 33], utilize tokenless pacers, while others adopt token-based pacers with pacing rates set at 1.25 times the estimated bandwidth to allow moderate bursts. Systems like Salsify [17] allow even higher burst rates (up to 5× the bandwidth) to minimize delays. Moreover, some latency-sensitive applications, such as cloud gaming, forgo pacing entirely, enabling unrestricted bursts.

However, these implementations share a common limitation: they **fix the level of burstiness to a certain extent** and lack adaptive sending pattern designs. These approaches were reasonable in previous contexts, particularly when RTTs dominated and pacing latency was not the primary concern. As discussed above, this is no longer sufficient. The straightforword choice between pacing and bursting can lead to excessive bursts that overwhelm the network or overly conservative pacing that unnecessarily delays frame packets at the sender. In §6, we will evaluate both bursty and paced sending methods.

(2) Failure to enable frame-level rate control while maintaining quality. CCAs strive to respond agilely to network variations, ensuring low latency and high throughput while mitigating congestion. Even with accurate bandwidth estimation, CC can only predict acceptable network bandwidth and relay this information to the encoder, lacking the ability to manage the variability in frame sizes produced by the encoder itself(§2). Thus, the latency induced by fluctuating frame sizes remains an unresolved challenge.

Some strategies aim to minimize frame size variability by directly controlling the encoder. For example, Salsify [17] re-encodes frames multiple times to comply with congestion window constraints, while other systems implement rate control mechanisms designed to enforce near-constant bitrate (CBR) encoding. However, these methods often achieve CBR by aggressively adjusting quantization parameters (QPs), leading to a deterioration in the quality of larger frames due to excessive lossy compression. Our evaluation(detailed in §6) further shows the significant quality loss of the accurate rate control solutions. This trade-off sacrifices visual quality to adhere to network constraints, which is particularly undesirable in high-quality aware real-time communication systems.



(a) P95 delay vs. Buffer Size (b) Loss rate vs. buffer size

Figure 10: Blindly transmitting bursts into the network is dangerous when the buffer size is unknown.

3.3 Design Challenges of ACE

The shortcomings of existing solutions highlight the need for a new approach to address pacing latency effectively. To this end, we propose ACE, which focuses on two key objectives: (i) **burstiness adaptive pacing** and (ii) **quality-preserving encoding control**. We face the following challenges in achieving these goals:

Dequeue: Allowing bursty sending and meanwhile minimizing packet loss risks. While pacing mechanisms have shortcomings, directly canceling pacing can be risky. Eliminating pacing means that burst packets are directly handled by the network. We conducted experiments by disabling WebRTC's pacing, sending encoded frames directly into the network. Using the emulation settings detailed in §6, we used Mahimahi [39] to emulate the network buffer size varied from 1000 packets (1500 Bytes MTU) downwards.

As shown in Fig. 10, when the buffer size decreases beyond a certain threshold, we observe a rapid increase in packet loss rate and a significant rise in video tail latency. At this point, the network buffer overflows, and the network cannot handle subsequent retransmission requirements, which is catastrophic to overall performance. However, with a sufficient buffer size, lower latency is observed compared to pacing, due to higher transient bandwidth utilization. From these observations, the key factor in determining whether to send a burst is the sufficiency of the network buffer. The sender method should satisfy (i) the capability to send frame-level bursts; and (ii) the ability to sense changes in the network buffer to prevent overflow. To address these requirements, we estimate network queuing size to detect changes in network buffers (§4.1) and use a conservative strategy to make sure the network is capable to handle bursts.

Enqueue: Achieving smoother frame sizes without compromising quality or adding overhead. As discussed in §2, frames vary in redundancy levels, and existing solutions often encode at a smooth bitrate, compromising quality. ACE, however, leverages the flexibility of encoding complexity to balance transmission delay and encoding times.

However, determining the optimal frame size based on coding complexity poses key challenges. The core goal is to select, from the available set of coding complexities, the one that minimizes total delay. First, a critical challenge arises from timing constraints: coding complexity must be specified before encoding begins, yet identifying whether a frame will be oversized can only be known after encoding. This necessitates accurate prediction of the encoded frame size prior to the encoding process. Second, we must ensure that the time invested in this trade-off is worthwhile, which requires modeling of the encoding time corresponding to each complexity level. To address these challenges, we define a gain function to evaluate the optimal coding complexity for each frame. We elaborate on the design details in §4.2.

4 Design

In this section, we first provide an overview of ACE design. In §4.1, we introduce ACE-N, the pacing controller. §4.2 presents ACE-C, the encoding controller.

ACE Workflow: ACE operates whenever the encoder starts processing a frame. ACE-N (§4.1) dynamically adjusting the bucket size of a token-based pacer based on the current observed network state. Meanwhile, ACE-C (§4.2) determines the coding complexity of the frame to be encoded by analyzing the raw input pixels.

It is worth noting that ACE does not interrupt the bitrate control logic in the existing delivery pipeline. Before encoded frames are produced, CCA monitors the network, predicting the available bandwidth and assessing how much data can be transmitted at any given moment. This information still serves as input for the encoder.

4.1 ACE-N: Pacing Controller

ACE-N sending controller manages the most appropriate sending patterns over short timescales observed from the network. First, it is crucial to emphasize the distinction between ACE-N and congestion control algorithms (CCAs). Since ACE-N takes network state as input and outputs how many packets are directly sent out, it is easy to confuse it with CCAs. However, a key difference lies in the context of RTC: the actual transmission rate is not strictly determined by the CCA. Instead, after processing by the encoder, the real output rate always fluctuates around the bitrate determined by the CCA, as discussed in detail in §2, resulting in transient bias from the target rate. In the long run, these encoded frames will eventually be transmitted, whether in bursts or at a paced rate. Thus, ACE-N focuses on solving the problem of *how* to send these already-encoded packets, rather than determining new bitrates. In this context, we assume that the bandwidth estimated by the CCA is accurate enough for long-term transmission; ACE-N, instead, focuses on whether small bursts will cause transient network overshoots. In summary, ACE-N operates in a different control space and involves distinct design considerations compared to CCAs, making them orthogonal. We also evaluate ACE over different CCAs in §6.6 to validate this.

The primary challenge for ACE-N is to balance between additional pacing delay (if the pacer is too conservative) and additional packet loss (if the pacer is too aggressive). Due to variability in frame sizes, oversized frames entering the pacer are more likely to incur extra pacing delays. While sending them into the network may mitigate pacing latency and benefit overall latency, a burst of frames can significantly increase the risk of packet loss, leading to further retransmission delays. The critical distinction lies in whether these frames will cause network buffer overflow. If overflow is not a concern, frames should be allowed to burst into the network; conversely, if overflow is likely, the sender should retain these packets.

To address these challenges, the design of the controller must fulfill two key requirements: First, it should manage frame-level data by controlling the burst size for individual frames. Second, it should respond to current network conditions, predicting the volume of data that can still be sent in bursts. For the first requirement, we employ a token-bucket-based pacer to enable finer-grained data transmission management within a frame. For the second, we utilize an adaptive bucket size management approach to react to the network changes.

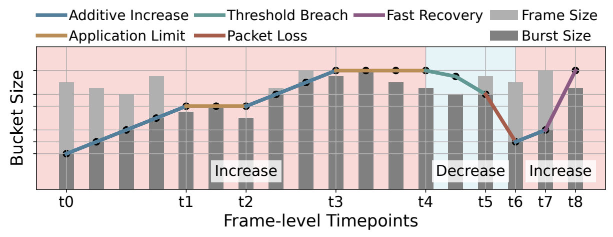


Figure 11: Bucket size adaption illustration

Token-bucket pacer structure. As a widely used traffic shaping mechanism [4, 47, 52], the token bucket accommodates bursty traffic while enforcing a defined average transfer rate. The algorithm operates by maintaining a metaphorical bucket that fills with tokens at a predetermined rate, with each token permitting the transmission of one packet. It is worth noting that we do not propose any new token bucket design – instead, we decide to use the existing token bucket filters for the convenience of implementation.

The token bucket is governed by two parameters: *Bucket Size* and *Token Rate*. The *Token Rate* controls how many packets to send on average. Recall that our design is orthogonal to CCAs and focuses only on the sending patterns within one RTT. Therefore, we set the Token Rate to be the sending rate determined by the CCA.

Bucket Size is the main focus of ACE-N. A larger *Bucket Size* allows for greater bursts of packets to be transmitted into the network. To handle bursty traffic, we adjust the *Bucket Size* dynamically, primarily in response to the observed network queue size. We will discuss in detail how ACE-N reacts to adjust the bucket size in the following analysis.

Adaptive bucket size. We explain how ACE-N adjusts the bucket size dynamically. To support this, estimating the in-network queue size is fundamental. Inspired by Copa [6], we estimate queue delay using RTT observations. The queue size is calculated by multiplying RTT with the current link capacity, which is determined using the widely-used PacketPair algorithm [25]. Based on the estimated queue size, ACE-N increases and decreases the bucket size as follows.

Increase. The token bucket increases its size when it believes the network can absorb a larger burst. However, it is fundamentally challenging to precisely estimate the bottleneck buffer capacity. Therefore, we have a two-fold design:

1. *Additive Increase:* When lacking historical information, we conservatively probe the available buffer size using an additive increase approach (Fig. 11: t0–t1, t2–t3, t6–t7).

2. *Fast Recovery:* We track two values: (i) the historical bucket size when the network buffer was empty, and (ii) the network queue size just before the most recent packet loss, scaled by a factor α ($0 < \alpha < 1$), as a conservative estimate of the bottleneck's maximum capacity (Fig. 11: t7–t8). The bucket size is updated by selecting the smaller of these values once queued network packets have cleared.

Such a design is to avoid aggressive overshooting while mitigating the slow growth of additive increase, which can cause unnecessary pacing delays. To further manage inaccuracies, we enforce an *application limit* (Fig. 11: t1–t2, t3–t4): if the bucket size exceeds the previous frame's size, no increase is applied to the current frame.

Decrease. While a larger bucket size can reduce pacing delays, it also increases the risk of packets overshooting the bottleneck buffer, leading to packet loss. To address this, we adopt two strategies to minimize packet loss:

1. *Queue-size-triggered* prevention: To proactively prevent packet loss, we define a threshold T to limit the number of packets queued in the network. If the predicted queue size exceeds T, the bucket size is reduced by the excess amount (Fig. 11, t4–t5). This threshold is empirically determined to balance reliability and performance: an excessively high threshold risks packet loss, while an overly low threshold leads to unnecessary pacing delays (For more details, see §6.5).

2. *Packet-loss-triggered* response: To reactively address the issue of packet loss, we respond immediately by halving the bucket size to avoid further losses (Fig. 11, t5–t6).

4.2 ACE-C: Encoding Controller

Table 1: Notations defined in ACE-C design

Symbol	Definition
ΔT_t	Reduction in transmission delay
ΔT_e	Increase in encoding delay
ΔF	Reduction in frame size
$\phi(c)$	Compression reduction factor for complexity c
F	Frame size
ρ	Actual proportion of frame size over average
$\hat{\rho}$	Predicted proportion of frame size over average
w	Weight used in SATD-based frame size prediction
offset	Offset used in SATD-based frame size prediction

ACE-C is designed to select the appropriate complexity-related encoding parameters for each frame. The symbols and definitions used can be found in Table 1.

As our objective is to trade encoding time for transmission time, the optimization goal of ACE-C can be expressed as

$$\text{Gain} = \Delta T_t - \Delta T_e = \frac{\Delta F}{\text{BWE}} - \Delta T_e \tag{1}$$

Here, ΔT_t, can be calculated by dividing the reduction in frame size by the available bandwidth, which we trust the CCA and estimate it as the estimated bandwidth (BWE). In this equation, ΔF can be replaced by $\phi(c) \cdot \rho \cdot \overline{F_n}$, where $\overline{F_n}$ is the average frame size over a short time window n.

As mentioned earlier, the target bitrate is tied to the BWE, and average bitrate coding allows frame sizes to fluctuate around the target bitrate within the time window. Thus $\text{BWE} = f \cdot \overline{F_n}$, where f is the frame rate. Now, given a complexity level $c \in \{c_0, c_1, \ldots, c_m\}$, we can compute the Gain as it relates to encoding complexity c:

$$\text{Gain}(c) = \frac{\rho \cdot \phi(c)}{f} - \Delta T_e(c) \tag{2}$$

Before encoding each frame, ACE-C selects the complexity c that maximizes the Gain. To do this, we must estimate the relative size ρ and the factors related to encoding complexity. We will explain these next.

SATD-based size prediction. This step predicts the relative frame size ρ compared to the target size, which should be done prior to encoding. ACE-C uses a simple yet effective method, based on the Sum of Absolute Transformed Differences (SATD), to estimate the size of the encoded frame. The SATD is computed by comparing the

Parameters	Range	Introduction
inter	[I8x8, I4x4, PSUB]	Inter Prediction Partitions
intra	[I8x8, I4x4]	Intra Prediction Partitions
i_me_method	[DIA, HEX]	Motion Estimation Method
i_subpel_refine	[1...4]	Subpixel motion estimation refinement level
i_trellis	True or False	Trellis Quantification

Table 2: x264 complexity parameter selection

frequency domain differences between the current frame and the preceding frame, making it a reliable proxy for the encoding process.

The SATD between two video frames F_n and F_{n-1} , denoted as S, can be expressed as:

$$\mathrm{S}(F_n,F_{n-1})=\sum_{x,y}|T(F_n(x,y))-T(F_{n-1}(x,y))| \tag{3}$$

where T is a transformation function, such as the discrete cosine transform (DCT) or the discrete Fourier transform (DFT), applied to each pixel of the frames. The absolute differences between transformed pixel values are summed across all pixels (x, y). In line with many rate control algorithms in video encoding, the relationship between SATD and bit rate is modeled linearly. Therefore, the predicted ρ is also proportional to the SATD, as follows:

$$\hat{\rho}=w\cdot\frac{\mathrm{S}}{\mathrm{S}_n}+\mathrm{offset} \tag{4}$$

Here, w and offset are initialized with empirical values and updated after encoding each frame.

Complexity Factors. For each encoding complexity level $c\in\{c_0,c_1,\ldots,c_m\}$, we need to estimate $\alpha(c)$ and $\Delta T_e(c)$ to calculate the Gain. These factors are initially set to empirical values and are updated based on actual frame size and encoding time.

Parameter Updating. The parameters w, offset, ϕ, and ΔT_e are updated using an exponentially weighted moving average (EWMA) after each new value is obtained, ensuring a smooth update process. We set α to 0.5 for this smoothing process:

$$P_t=\alpha P_t+(1-\alpha)P_{t-1} \tag{5}$$

In summary, before encoding each frame, ACE-C calculates the Gain for the available complexity options and selects the optimal complexity parameter. We implement and evaluate ACE-C using the libx264 encoder, which supports three complexity levels in the parameter set (detailed in §5.1).

5 Implementation

We introduce the implementation of ACE controller on WebRTC and x264 encoder.

5.1 x264 implementation

We have integrated our encoder controller design into x264 (Version 0.164.3191M), a widely used open-source H.264 software encoder. While the native WebRTC employs OpenH264 for H.264 encoding due to licensing constraints, we opted for x264 due to its more advanced rate control algorithm and greater flexibility in complexity management.

The x264 encoder offers three rate control methods: Constant Quantization Parameter (CQP), Constant Rate Factor (CRF), and Average Bit Rate (ABR). For real-time encoding, we utilize the recommended mode, ABR combined with the Variable Bitrate Video (VBV) buffer control. VBV serves as a hypothetical decoding buffer, allowing for frame size limitation and reduction of frame size volatility by adjusting the Quantization Parameter (QP). Our implementation is built upon the ABR + VBV modes, with the introduction of size prediction and complexity management.

Size Prediction. Fortunately, size prediction is already a feature of the x264 encoder for rate control, enabling us to leverage it without additional computational overhead. The encoder begins encoding a frame with a rate control analysis, measuring the content complexity against previous frames using the Sum of Absolute Transformed Differences (SATD). With the default complexity, the encoder predicts the ρ based on the SATD.

Complexity Management. We have identified five key coding complexity control parameters within the x264 encoder to effectively manage frame size, as outlined in Table 2. These parameters predominantly pertain to the partitioning and Motion Estimation(ME) phases. Utilizing these parameters, we have defined three distinct complexity modes, denoted as c_0, c_1, and c_2. Specifically, c_0 employs only 8x8 inter and intra partitions, DIA motion search, level 1 subpel refinement, and disables trellis quantification. Instead, c_2 includes all of the partition methods, the HEX motion search [38], level 4 subpel refinement, and enables trellis quantification. The difference between c_1 and c_2 is c_1 disables trellis quantification.

Interaction with Rate Control. In x264, the frame sizes are planned at the start of encoding a frame. Given that most frames are encoded with the base complexity mode c_0, the predicted frame size aligns with the size expected from c_0 complexity. However, when actual encoding commences, the encoder with higher complexity performs MB-level rate controls, adjusting the QP for each row to achieve the planned frame size. This approach can result in an original frame size with improved quality, which is not our intended outcome. To counter this, we modify the initial planned frame size using the preset α and the VBV budget, thereby reducing the frame size to achieve a similar quality level.

5.2 WebRTC implementation

We have implemented our controller design within the WebRTC C++ source code (based on Version M119). The native WebRTC framework utilizes a Pacer module to regulate the transmission rate. Our primary modifications involved enhancing the pacing module and integrating a Token Bucket control algorithm on top of it. We developed an ACE controller class to oversee the Token Bucket mechanism. The

Token Bucket rate is configured to match the estimated bitrate, and to bypass the Round-Trip Time (RTT) and packet reception timestamps from the RTCP receiver, we can predict the in-network buffer size.

The ACE controller adjusts the sending pace, overriding the Next Sending Time originally determined by the Pacing controller.

Interaction with CCA. We also made adaptations to the Google Congestion Control (GCC) algorithm used by WebRTC. GCC estimates the bandwidth partly through a delay-based controller. It employs an arrival-time filter, which is continuously updated based on the arrival times of incoming packets, calculated using the first packet of each packet group. WebRTC applies linear regression to predict the delay gradient trend, using a fixed number of packet groups within a window to determine the trendline. Packet groups are delineated by time intervals. The ACE design facilitates packet bursts, which reduces the number of packet groups. This reduction can make GCC less responsive to network changes, as it calculates the trendline over an extended period. To counteract this, we replaced the fixed number trendline estimator with a time-based window of 200 milliseconds for calculating the delay gradient.

6 Evaluation

In this section, we evaluate ACE with various experiments and results. For quick reference, we structure our analysis around the following questions:

Q1. What is the overall latency reduction achieved by ACE?
See §6.2, ACE cuts 95-th percentile latency by up to **43%** versus the state-of-the-art `WebRTC*` baseline.

Q2. Does ACE sacrifice video quality for lower latency?
See §6.2, **No**. ACE maintains the same VMAF score (perceptual quality) as the highest-quality baseline.

Q3. Which network conditions were tested?
See §6.1, most of our experiments were conducted in Mahimahi [39] emulation cover real-world **Wi-Fi**, **4G**, and **5G** traces from dataset in Zhuge [35]. We also evaluated in real-world campus Wi-Fi(§6.8) and production service(§6.9).

Q4. How does ACE perform on different content types?
See §6.2, ACE reduces latency by ∼**70%** on high-motion Gaming videos while preserving quality; gains are smaller on low-motion Lecture videos.

Q5. What is the impact on packet loss and other QoS metrics?
See §6.3, ACE keeps loss rates slightly above paced baselines but far below blind-burst schemes. We also present here other QoS metrics such as stall rate and end-to-end delays at different quantiles.

Q6. Does ACE interfere with existing CCAs?
See §6.6, **No**. Bandwidth-estimation accuracy and reaction to sudden drops remain unchanged for both GCC and BBR.

Q7. Is there additional CPU / memory overhead for ACE-C?
See §6.6, negligible; sender CPU rises by only ∼2 ms per frame and no extra load on the receiver.

Q8. How does ACE behave in real-world campus Wi-Fi?
See §6.8, on-campus tests confirm latency on par with low-latency baselines and the highest VMAF score.

Q9. Can ACE be implemented on production and what are the results?
See §6.9, trace-based experiments show ACE-N reduces latency by **15%**.

6.1 Main Experiment Setup

Testbed. We implement a testbed to automatically run the experiments and analyse the results. Experiments are conducted on a single server, where two separate processes initialize the sender and receiver. The receiver's downlink conditions are controlled using Mahimahi [39]—a network emulator that replays real network traces to simulate bandwidth variability. Network buffering is emulated via Mahimahi's drop-tail queue, with a fixed queue size of 100 Kilobytes across all experiments. During experiments, timestamps of frame transmission and reception (same time reference) are logged and used to compute end-to-end latency.

Video Selections. We chose 5 categories of videos to test, namely Music, Gaming, Sports, Vlog, and Lecture. We selected the most-watched video in the most-subscribed channel ranked and categorized at *HypeAuditor* Website [21]. The total duration of the videos is more than 1200 seconds, comparable to similar works [12, 14, 17]. The framerate of the videos is fixed at 30 fps.

Traces. We used real-world traces from Zhuge [35], which include both Wi-Fi and cellular network links. From this dataset, we sampled nine traces. Each trace consists of timestamps corresponding to available bandwidth, with a timestamp interval of 200 ms. The median bandwidth across all traces is 55 Mbps, with the 25th and 75th percentiles being 29 Mbps and 125 Mbps, respectively.

Competing Flows. To emulate the influence of in-network buffer change and evaluate fairness, we added competing flows in our experiments. We use Selenium [2] to randomly load Web pages from the Alexa Top-100 websites [1] through Google Chrome (Version 124.0.6367.201).

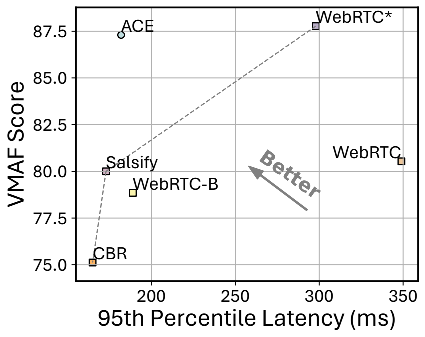
Metric. In the evaluation, we mainly focus on two metrics: (1) end-to-end delays, as a critical metric for RTC measured by various works [17, 37, 45] and (2) VMAF Score [44] introduced by Netflix to indicate perceptual quality. These two metrics are orthogonal to each other to reflect the overall performance.

Baselines. Our experiment encompasses several baselines for comparative analysis:

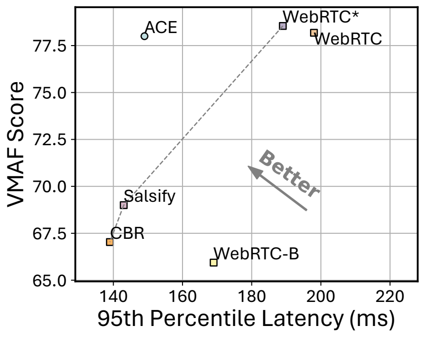
- `WebRTC.` This version utilizes the native WebRTC codes (Version M119) employed by Chrome, featuring a VP8 encoder.
- `WebRTC-B.` As a strawman solution, we test with a fixed but increased pacing rate to 2.5× of the BWE (a deprecated WebRTC setting before).
- `WebRTC*` (WebRTC + x264 Average Bitrate). We have integrated libx264 into WebRTC, and tuned it for zero-latency operation. The VBV (Video Buffering Verifier) setting follows [40].
- `CBR` (WebRTC + x264 Constant Bitrate). By adjusting the parameters of the encoder, we enable the generation of constant bitrates, precisely matching the target bitrate.
- `Salsify [17].` Salsify includes a functional VP8 codec and a congestion control mechanism tailored for real-time applications. It simultaneously generates two frames of different sizes, allowing the transmission protocol to select the most appropriate one.

6.2 Overall Performance

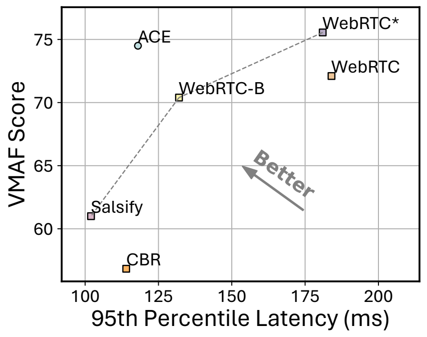
Fig. 12 illustrates the main performance trade-off plots between the 95th percentile latency and the average VMAF score across three sets of traces. Arrows to the upper left show the higher quality and lower latency.



(a) WI-FI trace



(b) 4G Cellular trace



(c) 5G Cellular trace

Figure 12: ACE performance over different network traces

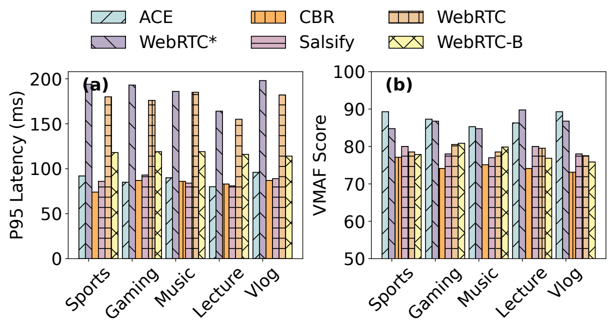
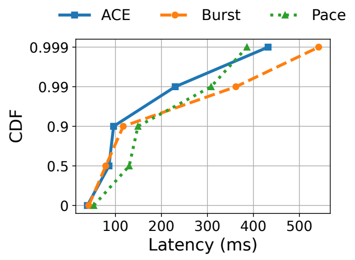


Figure 13: Comparison over video categories

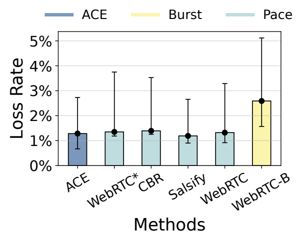
As shown in Fig. 12, WebRTC* achieves the highest quality among all baselines, largely due to its effective strategy for allocating bitrate across frames. However, it also incurs the highest latency, as its pacer introduces additional delays by smoothing frame bursts into the network. In contrast, CBR achieves the lowest latency across all traces by consistently producing frames that never exceed the instantaneous pacing rate. However, this comes at a significant cost to quality, with VMAF scores dropping by 7 to 15 points compared to WebRTC*, which demonstrates the quality loss of the lossy quantization methods. Other baselines balance different encoding rate control strategies and sending paces, positioning themselves along the trade-off frontier. ACE, however, breaks this trade-off by dynamically adjusting both encoding complexity and sending pace. As a result, it achieves a substantial P95 latency reduction of 34% compared to WebRTC* while maintaining the same level of video quality.

ACE outperforms all baselines among various network traces and videos. The performance across different network traces shows a consistent trend between Wi-Fi and Cellular traces, with ACE reliably outperforming all baselines. ACE has the highest VMAF Score among all the baselines, other baselines with comparable quality have to sacrifice from the increase of P95 latency by 43%, 23%, and 35% on Wi-Fi, 4G, and 5G traces, respectively. This indicates that ACE is robust to variations in network environments. In cellular traces, the performance gains are less pronounced. This is because sudden bandwidth drops occur more frequently in cellular networks, amplifying the contribution of congestion-related latency—thus reducing the relative impact of ACE 's optimizations.

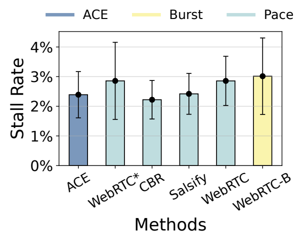
Results over different videos. We illustrate the comparative performance of ACE and baselines across various video categories in Fig. 13. ACE is capable of reducing latency by around 70% in Gaming videos while maintaining quality on par with WebRTC*, which delivers the highest quality across all video types. It is noteworthy that ACE's performance in terms of quality is similar to CBR, where CBR does not exhibit a significant quality loss compared to average bitrate encoding. This is particularly true for *Lecture* videos, which lack moving content, resulting in a relatively stable encoded frame size. This observation suggests that the ACE controller is better suited for deployment in scenarios with significant content variation, such as Virtual Reality (VR) streaming and Cloud Gaming.



(a) Latency CDF



(b) Loss rate



(c) Video stall rate

Figure 14: Other QoS metric

6.3 Other QoS Metric

We further analyze five other QoS metrics compared to the baselines.

Latency Distribution. We present the Cumulative Distribution Function (CDF) of latency in Fig. 14(a), comparing ACE with WebRTC* and WebRTC-B, marked as Pace and Burst, respectively. From the figure, it is evident that ACE consistently achieves the lowest latency across most percentiles. The Burst method exhibits similar latency to ACE around the 90th percentile but shows higher latency in the tail

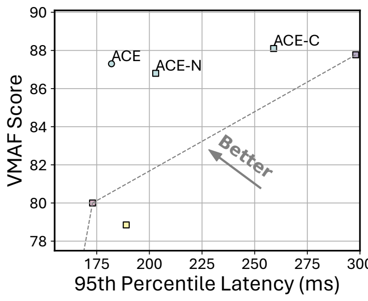


Figure 15: Ablation study

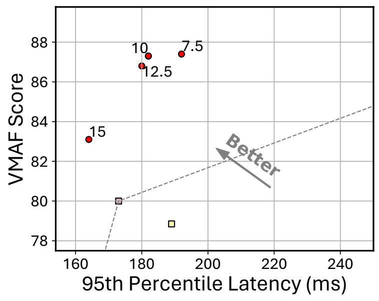


Figure 16: Sensitivity of the parameter T

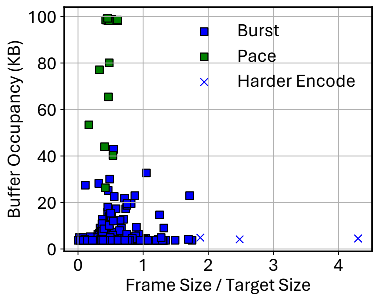


Figure 17: ACE decision understanding.

due to network overshooting. On the other hand, the Pace method smooths the transmission pattern but still results in higher latency across all percentiles compared to ACE, with pacing latency dominating. Notably, Pace achieves the lowest latency at the 99.9th percentile. We infer that despite ACE-N's controls, some false-negative decisions (allowing bursts when they should be paced) still cause network overshooting, contributing to ACE's slightly higher latency at this extreme percentile.

Loss Rate. We compare the loss rates between ACE and the other baselines. As shown in Fig. 14(b), baselines with bursty sending patterns exhibit loss rates exceeding 4%, significantly higher than those employing pacing mechanisms. ACE achieves a loss rate slightly higher than the Pacer method but remains well below the bursty baselines, around 1%. This demonstrates ACE 's ability to strike a balance between efficient transmission and maintaining low packet loss.

Frame Rate. Enabling frame dropping could introduce significant bias in video quality assessment. To ensure consistency and fairness across all measurements, we disabled the frame-dropping option. As a result, the average frame rates across all experiments are reported to be close to 30 FPS, consistent with the original video frame rate.

Video Stall Rate. We measured 100ms stall rate, which is defined as the ratio of the stall duration (periods where the video receiving interval exceeds 100 ms) to the total duration of a session. The stall rate is depicted in Fig. 14(c). ACE achieves an average video stall rate of 2.39%, which is among the lowest and is comparable to the best-performing baseline, `CBR`. Additionally, ACE reduces the stall rate by 16% compared to `WebRTC*` and 17% compared to `WebRTC-B`. These results demonstrate that ACE effectively mitigates video stalls caused by both pacing latency and retransmissions.

Link Utilization. We assess ACE's link utilization in comparison to the WebRTC default pacing method. Fig. 18 displays the CDF of the sending rate over a very fine timescale of 10 milliseconds, normalized by both bandwidth and estimated bandwidth. It demonstrates that ACE can maximize instantaneous link capacity utilization while avoiding overshoot of the bandwidth, unlike Pacing. Consequently, ACE optimizes link usage without causing excessive queuing in the network.

6.4 Ablation Study

In this experiment, we removed the new components implemented in ACE one by one to better understand their contribution to the total performance of the system. First, we removed the feature of ACE-N by sending the packets with a pacing rate fixed to estimate bandwidth. The performance degradation of this configuration is shown

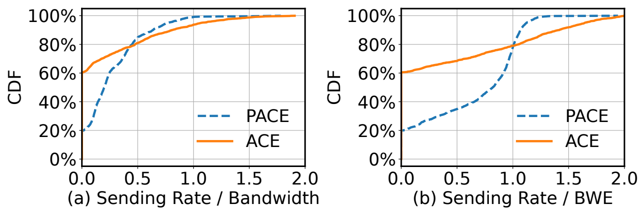


Figure 18: Link Utilization Analysis: ACE allows packets to be sent in bursts, resulting in longer silent periods, better utilizing the link by achieving a higher sending rate over short time scales.

in Fig. 15 as ACE-C; without this component, ACE has less latency reduction because the link is not well utilized on tiny timescales. However, only ACE-C still improves latency by smoothing the sending stream among frames, and the quality is higher by using a higher complexity. Next, we only keep ACE-N. The latency is significantly improved with the ACE-N, while the quality remains similar. Notably, the contribution of ACE-N is largely greater than ACE-C. However, both degradations lie on the upper left side of the envelope of baselines. It indicates that the two parts of our design can be employed separately and achieve a greater improvement when used together.

6.5 Parameter Sensitivity

We further evaluated the sensitivity of ACE parameters, focusing on the parameter T in decreasing the bucket-size, which is the threshold of packet size to trigger size reduction. We tested with T values of 7.5, 10, 12.5, and 15, and the results are shown in Fig. 16. Our findings indicate that ACE is not particularly sensitive to the parameter settings; all tested configurations outperformed the baseline envelope. Specifically, as T increases, there is a higher likelihood of packet loss, which can lead to a loss in video quality. However, by fully utilizing the link capacity, it is possible to achieve lower latency. The parameter allows operators to effectively balance quality and latency.

6.6 Microbenchmarks

Relative Size Prediction Accuracy. Fig. 19 shows the accuracy of the predicted relative size $\hat{\rho}$ using SATD. The color intensity indicates the probability distribution $P(\hat{\rho}|\rho)$. As shown, the predictions closely match the actual frame sizes, particularly in the higher range (oversized frames), where $\hat{\rho}$ accurately tracks ρ.

Interaction with Congestion Control. We evaluated the interaction between the ACE over two CCAs implemented on WebRTC – GCC and BBR [9]. Here, BBR is derived from WebRTC's legacy codebase [3]. Since ACE operates at a layer above CCA, our goal is to ensure that it does not interfere with CCA's effectiveness. The

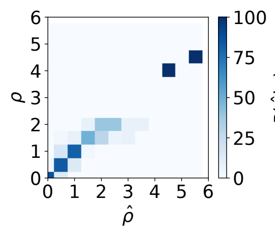


Figure 19: SATD precisely predicts the relative frame size.

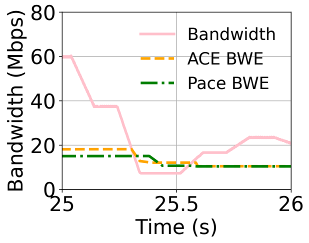


Figure 20: Reaction example to bandwidth drop.

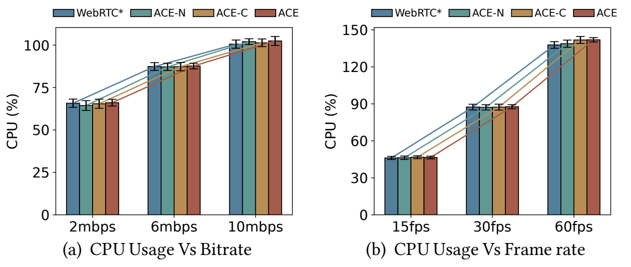

(a) CPU Usage Vs Bitrate (b) CPU Usage Vs Frame rate

Figure 22: Runtime CPU overhead on sender

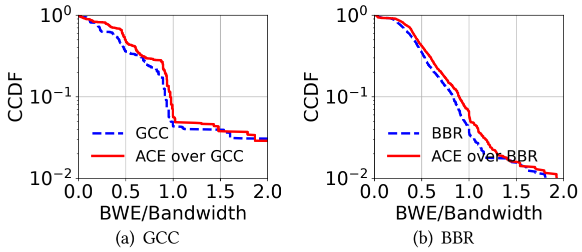

(a) GCC (b) BBR

Figure 21: Comparison of CC accuracy. BWE/Bandwidth < 1.0 indicates underestimation, otherwise overshoot.

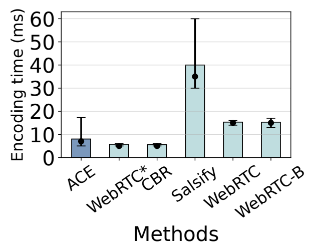


Figure 23: Encoding latency

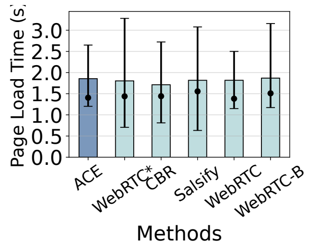


Figure 24: Page load times

impact of ACE is measured by the *accuracy of CCA's bandwidth estimation*, calculated as the ratio of estimated to actual bandwidth at 10 millisecond intervals. The CDF plot in Fig. 21(a) shows that the average bandwidth estimation accuracy of ACE is comparable to that of the pacing method, indicating no negative impact on both CCAs. Additionally, Fig. 20 illustrates how GCC responds to a sudden drop in bandwidth, with the reaction curves of ACE and the pacing method nearly overlapping, demonstrating similar behavior. Furthermore, to validate that ACE has no negative impact on CCAs in real-world scenarios, we evaluated its interaction with a custom CCA deployed in a production cloud gaming application. Detailed performance results are provided in §6.9.

Encoding Time. Fig. 23 shows the distribution of encoding times for various baselines compared to ACE. On average, ACE's encoding time is only 2 milliseconds higher than the x264 baseline, demonstrating its efficiency. Notably, the VP8 encoder exhibits higher encoding times than x264, while `Salsify` takes the longest due to its multiple encoding passes.

Runtime Overhead. We evaluated the overall system overhead. As illustrated in Fig.22, CPU usage and memory consumption on the sender increase with higher sending bitrates and frame rates. However, ACE introduces negligible additional overhead, with its impact being minimal compared to the influence of frame rate and bitrate. Additionally, we evaluated the overhead on the receiver, which remains similarly low. Detailed results and analysis are provided in the Appendix B.

Fairness. Given that our method sends streams unevenly on a small time scale, it is crucial to ensure that it does not negatively impact other streams. We measure fairness by assessing the page load time of competing streams, as shown in Fig. 24. Our method maintains an average level among the baselines, demonstrating that it does not significantly affect the competing flows.

6.7 ACE Deep Dive

Understanding ACE Decisions: Our example, depicted in Fig. 17, illustrates the ACE decision-making process for encoding and sending controllers across frames during the streaming of a gaming video. The x-axis shows frame sizes normalized relative to the target frame size, while the y-axis indicates buffer occupancy in the network buffer. The shape of each scatter point represents the joint decisions of ACE-N and ACE-C. As shown, most frames are transmitted in bursts (blue squares) and sent out completely. A minority of frames, which are more close to buffer overflow, are transmitted using pacing (green squares). For encoding, most frames use the c_0 complexity level; only oversized frames are processed with higher complexity (marked as X-shaped points). Clearly, ACE achieves smoothness through two key actions: encoding larger frames to reduce their size, and adjusting the burstiness of transmission based on network queuing conditions.

ACE-C Encoding Complexity Choices. In our tests, ACE-C offers three complexity levels (c_0 to c_2). We found that 97% of frames still use the default c_0, so ACE-C only steps in for the occasional large frame—for example, shown in Fig. 17, only for frames bigger than 1.6× the average size. For these outliers, the gain in Eq. 2 favors a higher level, trading extra encoding time for a worthwhile transmission delay. That's the reason the overall overhead for encoding stays small. Nevertheless, Fig. 15 shows ACE-C still trims the tail latency, confirming that these sporadic big frames are essential causes of high tail latency.

ACE-N Sending Pattern Decisions. Fig.25 illustrates the behavior of ACE-N under a typical network changing scenario in a 1-second time interval. At the start, the available bandwidth is underestimated by the BWE (blue line in Fig.25 (a)). ACE-N allows to send frames in a bursty pattern, as shown by the sharp spikes in buffer occupancy (red line in Fig.25 (b)). During this phase, the token bucket size (brown line in Fig.25 (c)) is large enough to accommodate bursts, preventing pacing delays by efficiently utilizing the underestimated bandwidth.

At t = 400 ms, the predicted queue size (green line in Fig.25(b)) exceeds the threshold T (dashed line). To avoid triggering the drop-tail

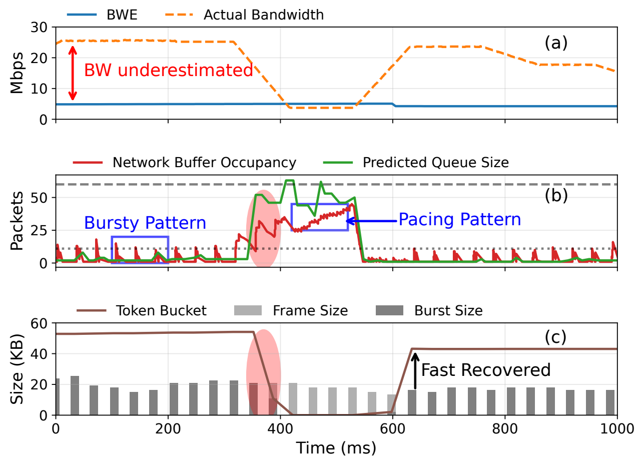


Figure 25: Understanding of ACE-N adaptive pacing

limit of the bottleneck buffer, ACE-N reduces the token bucket size. This transition effectively switches the sending pattern from bursty to pacing, as shown by the smoother network buffer occupancy after the adjustment. By responding to the predicted queue size, ACE-N avoids network overshooting and packet loss.

At t = 600 ms, the predicted queue size drops to zero, indicating that the network buffer is no longer occupied. In response, ACE-N allows the fast recovery mechanism, quickly restoring the token bucket size to enable larger bursts (Fig.25(c)). This is the end of an increase/decrease cycle.

6.8 Real-World Experiment

To further validate the performance, we conducted experiments in a real-world network environment. The sender was a Ubuntu server located in the university's data center running ACE over WebRTC, while a Windows laptop connected to the campus wireless network (eduroam) acted as the receiver. For comparison, we included baselines in the Wi-Fi trace emulation at the frontier (`WebRTC*`, `CBR`, and `Salsify` Additionally, we tested `Google Meet` as a representative industry-standard baseline to reflect current real-world performance.

The evaluation involved transmitting a high-motion gaming video from the sender to the receiver, lasting approximately 200 seconds. Each frame of the video contained QR codes for precise identification. The receiver captured the screen and analyzed the metrics, focusing on end-to-end latency and VMAF score—consistent with the metrics used in the emulation environment. The tests were conducted over a 24-hour period, covering both peak and off-peak hours on campus. Fig. 26 presents the CDF of end-to-end latency and VMAF score across all test scenarios.

The results demonstrated that ACE delivered consistently low latency, comparable to `CBR` and `Salsify`, but stood out by achieving the highest VMAF score, on par with `WebRTC*`, while significantly reducing latency. Notably, Salsify's performance was constrained by extended encoding times, leading us to lower the resolution to 540p for real-time encoding, which caused a significant quality drop, reflected in a median VMAF score below 60. In contrast, `Google Meet` maintained a stable VMAF score of around 66, but its performance was likely optimized for video conferencing rather than high-motion content, making it less suitable for gaming scenarios. These findings highlight ACE 's strong real-world performance, successfully achieving low latency and high video quality.

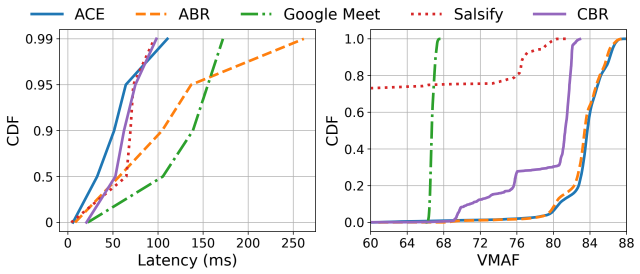


Figure 26: Real-world experiment results

Method	Stall Rate	Latency	Recv FPS
ACE-N	**2.89**	**136.97**	**56.80**
AlwaysPace	2.96	161.06	56.58
AlwaysBurst	13.37	323.34	53.79

Table 3: Performance on production application

6.9 Production Application Experiment

In this section, we incorporate ACE-N into the production RTC engine of our cloud gaming application, which serves millions of users nationwide. To validate the effectiveness of ACE-N, we benchmark it against the *AlwaysPace* and *AlwaysBurst* baselines using real-world network traces collected from weak network environments such as canteens, coffee shops, and airports. Each method was evaluated over a period of 1333 seconds using a 60 FPS game content video as the test case. We report stall rate, average latency, and received frame rate (Recv FPS) as the primary performance metrics, which directly impact the quality of user experience.

As shown in Table 3, ACE-N demonstrates significant improvements over the baselines across all key performance metrics. Compared to *AlwaysPace*, ACE-N reduces latency by 15.0%, while maintaining a higher received frame rate (56.80 vs. 56.58 FPS). Against *AlwaysBurst*, ACE-N delivers a dramatic reduction in stall rate (2.89 vs. 13.37) and latency (136.97 ms vs. 323.34 ms), alongside a 5.6% improvement in received frame rate (56.80 vs. 53.79 FPS). These results highlight ACE-N's ability to adapt effectively to various network conditions, ensuring a smoother gaming experience with lower latency. Notably, the *AlwaysBurst* baseline suffered from extremely high packet loss under weak network conditions. The aggressive bursting behavior of this approach exacerbates retransmissions, significantly increasing latency and degrading overall performance. In contrast, ACE-N mitigates these issues by dynamically adjusting its sending patterns. By carefully balancing bursty sending to reduce latency while avoiding excessive packet loss, ACE-N effectively overcomes the limitation of traditional burst-or-pace methods and ensures optimal performance.

Ethical Concerns: We only collect the objective measurement logs of the participating volunteers to obtain the timestamps of all received frames. Other metrics are calculated based on that. This data collection process does not involve any user privacy information and has already obtained the consent of the users.

7 Discussions

Scenario Limitations. As evaluated in §6.2, ACE is particularly well-suited for scenarios with high-quality, rapidly changing content—such as cloud gaming and VR streaming—due to its ability to adapt to fluctuating frame sizes. However, in use cases where frame sizes remain relatively stable (e.g., video conferencing), ACE provides marginal benefits over constant bitrate (CBR) baselines. In practice, our real-world tests of ACE and baseline methods on static content showed comparable quality levels to the baselines.

Encoder Generalization. In our evaluation, we implemented the ACE encoder using the widely-adopted software encoder x264, which is an H.264 encoder offering a comprehensive set of tools to control encoding complexity. In contrast, the default WebRTC H.264 encoder, OpenH264, lacks such a broad range of tools. In Appendix A, we detail how our encoding complexity control mechanisms can be adapted to other mainstream video encoders (e.g., AV1 and HEVC), demonstrating the generalization of our method across mainstream encoder architectures. Notably, ACE-C is currently limited to software encoders as the complexity is typically fixed for hardware encoding. We therefore aim to share these insights with the hardware encoding community to advocate for more granular interfaces for controlling the encoding process.

8 Related work

Real-time Video Coding. Various methods have been developed in video coding to optimize real-time communication (RTC) performance. A primary focus has been enhancing encoder resilience to transmission errors and packet losses, including through Scalable Video Coding (SVC) [15, 48, 49], error concealment techniques [26, 54, 62], and joint source-channel coding [8, 13, 28]. Another notable trend includes neural codec adaptations in RTC systems [10, 12, 50].

The mismatch between video codecs and network, as a key motivation of this paper, was firstly studied very early when people noticed that I-frames can overshoot network and fixed by intra-block refreshing [27]. Recent work also addresses the mismatch through cross-layer designs [17, 24, 60, 61]. Notably, these co-designs are still fundamentally quality-latency trade-offs.

Transport-layer innovations. Numerous prior efforts have focused on optimizing latency for RTC in the transport layer. The first and most studied one is through low-latency congestion control [6, 24, 45, 53, 57]. Another function of the transport layer is reliable transmission; recent researches include optimizations in Forward Error Correction (FEC) [5, 11, 30, 46] and retransmission improvements [7, 36, 41]. However, to our knowledge, adaptive strategies via pacing mechanisms are understudied.

Our approach is orthogonal to other transport-layer strategies: while CCAs determine how many packets to send within one RTT, we optimize how those packets are transmitted within that interval. ACE can be readily integrated with existing CCAs to enhance performance. The orthogonality also works for reliable transmission. That said, our strategy ACE-N takes loss as input; random loss which should be dealt with by FEC may be noise to our algorithm. We leave co-designing ACE with loss recovery mechanisms for our future work.

9 Conclusion

ACE introduces a burstiness control framework designed for high-quality real-time communication, addressing the mismatch between encoding and transmission patterns. By dynamically adjusting encoding complexity at the frame level to smooth frame sizes (ACE-C) and adaptively managing transmission patterns to mitigate pacing delays (ACE-N), ACE significantly reduces end-to-end latency without sacrificing quality. Experimental results demonstrate that ACE achieves up to a 43% reduction in 95th-percentile latency compared to state-of-the-art solutions, all while maintaining leading visual quality (VMAF). ACE aims to provide insights into managing transient sending patterns and leveraging adaptive encoding complexity for real-time transmission.

Acknowledgements. We sincerely thank our shepherd Dongsu Han, anonymous reviewers, and labmates in SPARKLab from HKUST for their valuable feedback. This work was supported by Guangdong NSF Project (No. 2025A1515010460). Zili Meng is the corresponding author.

References

[1] 2022. Alexa Top Websites >> ExpiredDomains.net. https://member.expireddomains.net/domains/researchalexamillion/.
[2] 2022. Selenium. https://www.selenium.dev/.
[3] Aerys Nan. 2021. BBR Development - Groups Google . https://groups.google.com/g/bbr-dev/c/1EPG5UwBANo.
[4] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2024. Towards provably performant congestion control. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 951–978. https://www.usenix.org/conference/nsdi24/presentation/agarwal-anup
[5] Congkai An, Huanhuan Zhang, Shibo Wang, Jingyang Kang, Anfu Zhou, Liang Liu, Huadong Ma, Zili Meng, Delei Ma, Yusheng Dong, and Xiaogang Lei. 2025. Tooth: Toward Optimal Balance of Video QoE and Redundancy Cost by Fine-Grained FEC in Cloud Gaming Streaming. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 635–651. https://www.usenix.org/conference/nsdi25/presentation/an
[6] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical {Delay-Based} congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 329–342.
[7] Ethan Blanton, Vern Paxson, and Mark Allman. 2009. *TCP Congestion Control*. RFC 5681. IETF.
[8] Eirina Bourtsoulatze, David Burth Kurka, and Deniz Gündüz. 2019. Deep joint source-channel coding for wireless image transmission. *IEEE Transactions on Cognitive Communications and Networking* 5, 3 (2019), 567–579.
[9] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue* 14, 5 (2016), 20–53.
[10] Bo Chen, Zhisheng Yan, Yinjie Zhang, Zhe Yang, and Klara Nahrstedt. 2024. LiFteR: Unleash Learned Codecs in Video Streaming with Loose Frame Referencing. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 533–548. https://www.usenix.org/conference/nsdi24/presentation/chen-bo
[11] Sheng Cheng, Han Hu, and Xinggong Zhang. 2023. ABRF: Adaptive BitRate-FEC joint control for real-time video streaming. *IEEE Transactions on Circuits and Systems for Video Technology* 33, 9 (2023), 5212–5226.
[12] Yihua Cheng, Ziyi Zhang, Hanchen Li, Anton Arapin, Yue Zhang, Qizheng Zhang, Yuhan Liu, Kuntai Du, Xu Zhang, Francis Y Yan, et al. 2024. {GRACE}:{Loss-Resilient}{Real-Time} Video through Neural Codecs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 509–531.
[13] Kristy Choi, Kedar Tatwawadi, Aditya Grover, Tsachy Weissman, and Stefano Ermon. 2019. Neural Joint Source-Channel Coding. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 1182–1192. https://proceedings.mlr.press/v97/choi19a.html
[14] Mauro Conti, Simone Milani, Ehsan Nowroozi, and Gabriele Orazi. 2021. Do not deceive your employer with a virtual background: A video conferencing manipulation-detection system. *arXiv preprint arXiv:2106.15130* (2021).
[15] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. 2022. Swift: Adaptive Video Streaming with Layered Neural Codecs. In *19th USENIX Symposium on Networked Systems Design*

and Implementation (NSDI 22). USENIX Association, Renton, WA, 103–118. https://www.usenix.org/conference/nsdi22/presentation/dasari
[16] Sandesh Dhawaskar Sathyanarayana, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. 2023. Converge: Qoe-driven multipath video conferencing over webrtc. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 637–653.
[17] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. 2018. Salsify:{Low-Latency} network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 267–282.
[18] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi
[19] Google. 2021. Paced Sending. https://chromium.googlesource.com/external/webrtc/+/master/modules/pacing/g3doc/index.md.
[20] HandBrake Team. 2023. HandBrake Documentation — Constant Quality vs Average Bit Rate. https://handbrake.fr/docs/en/latest/technical/video-cq-vs-abr.html [Online; accessed 19-Sep-2024].
[21] HypeAudio. 2021. Top YouTube Channels | HypeAuditor YouTube Ranking. https://hypeauditor.com/top-youtube/.
[22] ITU/ISO/IEC. 2018. HEVC Test Model (HM) Documentation. http://hevc.info/HM-doc/.
[23] Bart Jansen, Timothy Goodwin, Varun Gupta, Fernando Kuipers, and Gil Zussman. 2018. Performance Evaluation of WebRTC-based Video Conferencing. *SIGMETRICS Perform. Eval. Rev.* 45, 3 (March 2018), 56–68. https://doi.org/10.1145/3199524.3199534
[24] Zhidong Jia, Yihang Zhang, Qingyang Li, and Xinggong Zhang. 2024. Tackling Bit-Rate Variation of RTC Through Frame-Bursting Congestion Control. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*. 1–11. https://doi.org/10.1109/ICNP61940.2024.10858541
[25] Srinivasan Keshav. 1995. Packet-pair flow control. *IEEE/ACM transactions on Networking* (1995), 1–45.
[26] Vineeth Shetty Kolkeri. 2009. Error concealment techniques in H. 264/AVC, for video transmission over wireless networks. (2009).
[27] Sunil Kumar, Liyang Xu, Mrinal K Mandal, and Sethuraman Panchanathan. 2006. Error resiliency schemes in H. 264/AVC standard. *Journal of Visual Communication and Image Representation* 17, 2 (2006), 425–450.
[28] David Burth Kurka and Deniz Gündüz. 2020. Deepjscc-f: Deep joint source-channel coding of images with feedback. *IEEE Journal on Selected Areas in Information Theory* 1, 1 (2020), 178–193.
[29] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*. 183–196.
[30] Insoo Lee, Seyeon Kim, Sandesh Sathyanarayana, Kyungmin Bin, Song Chong, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. 2022. R-fec: Rl-based fec adjustment for better qoe in webrtc. In *Proceedings of the 30th ACM International Conference on Multimedia*. 2948–2956.
[31] Jinsung Lee, Sungyong Lee, Jongyun Lee, Sandesh Dhawaskar Sathyanarayana, Hyoyoung Lim, Jihoon Lee, Xiaoqing Zhu, Sangeeta Ramakrishnan, Dirk Grunwald, Kyunghan Lee, and Sangtae Ha. 2020. PERCEIVE: deep learning-based cellular uplink prediction using real-time scheduling patterns. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services* (Toronto, Ontario, Canada) *(MobiSys '20)*. Association for Computing Machinery, New York, NY, USA, 377–390. https://doi.org/10.1145/3386901.3388911
[32] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. 2020. Tack: Improving wireless transport performance by taming acknowledgments. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 15–30.
[33] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. Same standards, different decisions: A study of QUIC and HTTP/3 implementation diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. 14–20.
[34] Zili Meng, Nirav Atre, Mingwei Xu, Justine Sherry, and Maria Apostolaki. 2024. Confucius: Achieving Consistent Low Latency with Practical Queue Management for Real-Time Communications. arXiv:2310.18030 [cs.NI] https://arxiv.org/abs/2310.18030
[35] 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*. 193–206.
[36] Zili Meng, Xiao Kong, Jing Chen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. 2024. Hairpin: Rethinking packet loss recovery in edge-based interactive video streaming. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 907–926.
[37] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. 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.
[38] Loren Merritt and Rahul Vanam. 2007. Improved Rate Control and Motion Estimation for H.264 Encoder. In *2007 IEEE International Conference on Image Processing*, Vol. 5. V – 309–V – 312. https://doi.org/10.1109/ICIP.2007.4379827
[39] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 417–429. https://www.usenix.org/conference/atc15/technical-session/presentation/netravali
[40] Nvidia. 2024. NVENC Video Encoder API Programming Guide - NVIDIA Docs. https://docs.nvidia.com/video-technologies/video-codec-sdk/12.1/nvenc-video-encoder-api-prog-guide/index.html
[41] Colin Perkins, Orion Hodson, and Vicky Hardman. 1998. A survey of packet loss recovery techniques for streaming audio. *IEEE network* 12, 5 (1998), 40–48.
[42] Peter Holslin. 2024. The Best Internet for VR Gaming, Streaming, and More. https://www.highspeedinternet.com/resources/best-internet-for-vr. Accessed: 2024-09-12.
[43] Yi Qiao, Han Zhang, and Jilong Wang. 2024. NetFEC: In-network FEC Encoding Acceleration for Latency-sensitive Multimedia Applications. In *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*. 2348–2357. https://doi.org/10.1109/INFOCOM52122.2024.10621183
[44] Reza Rassool. 2017. VMAF reproducibility: Validating a perceptual practical video quality metric. In *2017 IEEE international symposium on broadband multimedia systems and broadcasting (BMSB)*. IEEE, 1–2.
[45] Devdeep Ray, Connor Smith, Teng Wei, David Chu, and Srinivasan Seshan. 2022. SQP: Congestion Control for Low-Latency Interactive Video Streaming. arXiv:2207.11857 [cs.NI] https://arxiv.org/abs/2207.11857
[46] Michael Rudow, Francis Y Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and KV Rashmi. 2023. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 953–971.
[47] Sambit Sahu, Philippe Nain, Christophe Diot, Victor Firoiu, and Don Towsley. 2000. On achievable service differentiation with token bucket marking for TCP. *ACM SIGMETRICS Performance Evaluation Review* 28, 1 (2000), 23–33.
[48] Thomas Schierl, Thomas Stockhammer, and Thomas Wiegand. 2007. Mobile video transmission using scalable video coding. *IEEE transactions on circuits and systems for video technology* 17, 9 (2007), 1204–1217.
[49] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. 2007. Overview of the scalable video coding extension of the H. 264/AVC standard. *IEEE Transactions on circuits and systems for video technology* 17, 9 (2007), 1103–1120.
[50] Vibhaalakshmi Sivaraman, Pantea Karimi, Vedantha Venkatapathy, Mehrdad Khani, Sadjad Fouladi, Mohammad Alizadeh, Frédo Durand, and Vivienne Sze. 2024. Gemino: Practical and Robust Neural Compression for Video Conferencing. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 569–590. https://www.usenix.org/conference/nsdi24/presentation/sivaraman
[51] Bruce Spang, Shravya Kunamalla, Renata Teixeira, Te-Yuan Huang, Grenville Armitage, Ramesh Johari, and Nick McKeown. 2023. Sammy: smoothing video traffic to be a friendly internet neighbor. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 754–768.
[52] Puqi Perry Tang and T-YC Tai. 1999. Network traffic characterization using token bucket model. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, Vol. 1. IEEE, 51–62.
[53] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, et al. 2024. Pudica: Toward {Near-Zero} Queuing Delay in Congestion Control for Cloud Gaming. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 113–129.
[54] Yi Wang, Xiaoqiang Guo, Feng Ye, Aidong Men, and Bo Yang. 2013. A novel temporal error concealment framework in H. 264/AVC. In *2013 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 1–6.
[55] Yilin Wang, Sasi Inguva, and Balu Adsumilli. 2019. YouTube UGC Dataset for Video Compression Research. In *2019 IEEE 21st International Workshop on Multimedia Signal Processing (MMSP)*. 1–5. https://doi.org/10.1109/MMSP.2019.8901772
[56] Wikipedia contributors. 2023. Constant bitrate — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Constant_bitrate [Online; accessed 19-Sep-2024].
[57] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. 2013. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 459–471. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/winstein
[58] x265 Project. 2018. x265 Command Line Options. https://x265.readthedocs.io/en/master/cli.html.

[59] Xiaokun Xu and Mark Claypool. 2022. Measurement of cloud-based game streaming system response to competing TCP cubic or TCP BBR flows. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 305–316.

[60] Zicheng Zhang, Hao Chen, Xun Cao, and Zhan Ma. 2023. Anableps: Adapting bitrate for real-time communication using VBR-encoded video. In *2023 IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 1685–1690.

[61] Anfu Zhou, Huanhuan Zhang, Guangyuan Su, Leilei Wu, Ruoxuan Ma, Zhen Meng, Xinyu Zhang, Xiufeng Xie, Huadong Ma, and Xiaojiang Chen. 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.

[62] Jie Zhou, Bo Yan, and Hamid Gharavi. 2010. Efficient motion vector interpolation for error concealment of H. 264/AVC. *IEEE Transactions on Broadcasting* 57, 1 (2010), 75–80.

[63] Yuhan Zhou, Tingfeng Wang, Liying Wang, Nian Wen, Rui Han, Jing Wang, Chenglei Wu, Jiafeng Chen, Longwei Jiang, Shibo Wang, Honghao Liu, and Chenren Xu. 2024. AUGUR: Practical Mobile Multipath Transport Service for Low Tail Latency in Real-Time Streaming. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. USENIX Association, Santa Clara, CA, 1901–1916. https://www.usenix.org/conference/nsdi24/presentation/zhou-yuhan

A Encoding Complexity Parameters

In this section, we share empirical insights into selecting encoding complexity parameters across different codecs.

A.1 Why Choose the x264 Encoder?

We selected the x264 encoder for its optimized performance and suitability for real-time encoding. As a widely adopted industry standard, x264 supports a broad range of encoding latencies—from 5 ms to 200 ms per frame in our tests with different presets—offering flexibility to balance encoding and transmission time. Importantly, x264 is not the only encoder compatible with our design: other mainstream codecs such as HEVC and AV1 share a similar architectural framework to H.264.

A.2 Mainstream Codecs Overview

Mainstream codecs include Google's VPX series and JVET's H.26x series. These codecs share a common framework encompassing prediction, transformation, and quantization: they leverage intra and inter-frame prediction to reduce temporal redundancy, transform coding like DCT to reduce spatial redundancy, and apply quantization for lossy compression. Additionally, they provide extensive parameters to control encoding complexity and quality.

A.3 Parameters for HEVC (H.265)

HEVC, also known as H.265, was developed by JVET as the successor to H.264, offering superior compression efficiency for high-resolution content. The reference software, HM [22], only supports standard-compliant coding tools that actually operate at the highest complexity level of HEVC. To enable adaptive complexity control for HEVC, a viable approach is to use x265— a widely adopted alternative encoder. Parameter selection for x265 is largely analogous to its predecessor x264: operators can easily adjust complexity via **presets**. Notably, our experience tuning x265 shows that presets beyond **medium** yield minimal gains in encoding complexity while incurring significant increases in encoding time/overhead. Thus, we recommend controlling complexity via partition levels, specifically using the `min-cu-size` parameter [58].

A.4 VP9 and AV1

VP9 and AV1—developed by Google—are widely used in web streaming platforms such as YouTube) due to their open-source, royalty-free nature. Both codecs provide adjustable parameters via a **speed** control, which influences encoding complexity and output quality: higher speed settings reduce encoding time by simplifying processes like motion estimation and intra prediction. However, unlike x264/x265, the Google-developed encoders for these codecs do not allow frame size to be significantly adjusted via *speed* alone. Thus, we also suggest the adjustment of block division parameters. For example, AV1 supports *superblock* size configurations (from 128×128 to 64×64) in its sequence header, which can be tuned to regulate complexity.

B Overhead Experiment Results

B.1 Runtime Overhead vs. Encoding Complexity

Beyond measuring encoding/decoding time across different encoding complexities (Fig. 5), we also evaluate runtime CPU and memory overhead. For CPU overhead (Fig. 27), we observe that as complexity increases, the sender's overhead rises significantly while

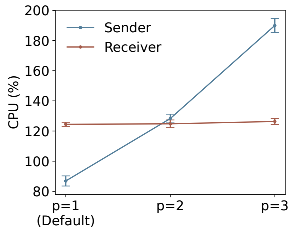
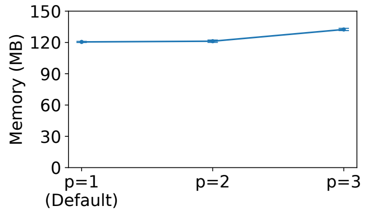
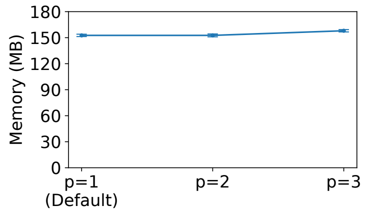


Figure 27: Runtime CPU Overhead vs. Encoding Complexity

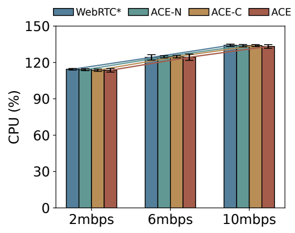


(a) Sender

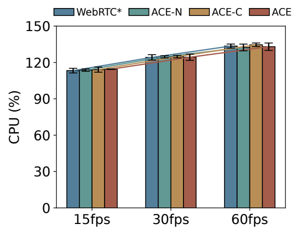


(b) Receiver

Figure 28: Runtime Memory vs. Encoding Complexity

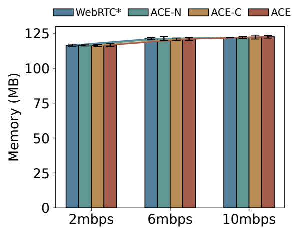


(a) CPU Usage vs. Bitrate

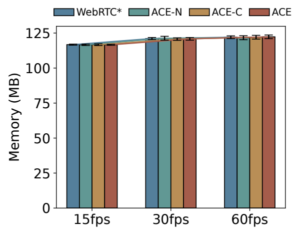


(b) CPU Usage vs. Frame Rate

Figure 29: Runtime CPU Overhead on Receiver



(a) Memory vs. Bitrate

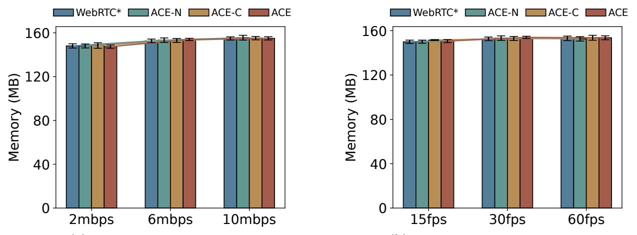


(b) Memory vs. Frame Rate

Figure 30: Runtime Memory Overhead on Sender

Algorithm 1: Logic for Adaptive Bucket Size

```
Input: Historical bucket size, predicted queue size, $\alpha$, application limit, threshold
Output: Updated bucket size
Initialization: Set initial bucket size $B$;
Procedure Increase(B)
    if Historical information is unavailable then
        /* Additive Increase */;
        $B \leftarrow B + \Delta$;
    end
    else
        /* Fast Recovery */;
        $B_{\text{historical}} \leftarrow$ bucket size when buffer was empty;
        $B_{\text{recent}} \leftarrow$ queue size just before packet loss;
        $B \leftarrow \min(B_{\text{historical}}, \alpha \cdot B_{\text{recent}})$;
    end
    /* Application Limit */;
    if Bucket size exceeds previous frame's size then
        $B \leftarrow B$;
        /* No change */
    end
Procedure Decrease(B)
    if Predicted queue size exceeds threshold then
        /* Threshold-Breach-Triggered Prevention */;
        $B \leftarrow B -$ (Predicted Queue Size $-$ Threshold);
    end
    if Packet loss is detected then
        /* Packet-Loss-Triggered Response */;
        $B \leftarrow B/2$;
    end
```



(a) Memory vs. Bitrate (b) Memory vs. Frame Rate

Figure 31: Runtime Memory Overhead on Receiver

the receiver’s remains largely unchanged, mirroring the trends in encoding/decoding time. For memory overhead (Fig. 28(b)), both sender and receiver exhibit minimal increases with rising complexity, though the sender’s growth is slightly greater, further confirming that the receiver’s overhead remains relatively stable across complexities.

B.2 Runtime Overhead Evaluation

In addition to measuring CPU overhead on the sender (Fig. 22), we also evaluate it on the receiver (Fig. 29). Despite ACE dynamically adjusting encoding complexity, the results indicate no increase in CPU overhead on the receiver compared to the original `WebRTC`. Notably, CPU overhead on both sender and receiver is heavily influenced by encoding bitrates and frame rates. Similarly, memory overhead measurements (Fig. s 30 and 31) show that ACE introduces no additional impact on either side.

C Pseudocode of ACE-N

Algorithm 1 shows how ACE-N’s bucket size adaptation logic works according to different conditions.