Real-Time Video Gets a Fast Lane via Smart Queue Flushing at the Wireless Edge

Yixuan Zhang[1,3], Zili Meng[2], Enhuan Dong[1], Yan Zhang[1,3], Jia Zhang[4], Mingwei Xu[3,1], Jianping Wu[1]
[1]Tsinghua University
[2]Hong Kong University of Science and Technology
[3]Quan Cheng Laboratory
[4]Zhongguancun Laboratory

IEEE INFOCOM 2026 - IEEE Conference on Computer Communications | 979-8-3315-4961-9/26/$31.00 ©2026 IEEE | DOI: 10.1109/INFOCOM59046.2026.11571420

***Abstract*—Real-time communication (RTC) applications demand not only low average latency but also tight tail-delay bounds to ensure smooth user experience. However, sudden fluctuations in wireless networks can cause in-flight packets to accumulate in bottleneck queues, delaying or invalidating subsequent frames. Traditional mechanisms focus on rate adaptation but largely overlook managing already enqueued packets that contribute to tail delay. We present Gecko, a lightweight end-to-network coordination mechanism that enables frame-aware queue flushing without requiring any in-network packet modification or protocol negotiation. Gecko-enabled routers monitor queuing delay and implicitly signal the sender, which then makes frame-skipping decisions and conveys flushing intent through minimal in-band RTP markings. This approach preserves end-to-end integrity and is broadly compatible with existing RTC applications. We evaluate Gecko via trace-driven simulations and real-world experiments. Results show that Gecko reduces average frame delay by 21.8% and cuts tail-delay frame ratios by 25% to 91%, demonstrating both its effectiveness and deployability in wireless RTC environments.**

I. Introduction

Real-time communication (RTC) applications such as video conferencing, cloud gaming, teleoperation, and virtual reality are increasingly prevalent in today's Internet [31], [10]. These applications demand not only low end-to-end latency but also strict consistency at extreme tail percentiles (e.g., 99.9th or higher) to preserve user experience [27], [47]. However, the best-effort nature of the Internet, which provides no intrinsic latency guarantees, makes achieving this goal fundamentally challenging. The problem is exacerbated in wireless networks like WiFi and cellular, where frequent delay jitter, bandwidth fluctuations, and transient losses lead to persistent stalls and degraded application quality. (§II-A)

This problem has been widely studied across multiple protocol layers. Numerous techniques exist: application-layer redundancy and adaptive coding [36], delay-sensitive congestion control [3], [18], in-network active queue management (AQM) [2], and cross-layer feedback schemes [27], [23]. Yet despite decades of effort, RTC applications still suffer from poor tail performance in practice. Recent measurement studies show that platforms like Zoom and Webex continue to exhibit frequent frame stalls, especially under fluctuating wireless network conditions [7], [29].

We argue that a fundamental limitation of existing approaches is their inability to manage in-flight packets during the control loop reaction time. When network capacity suddenly degrades, existing end-to-end schemes, no matter how responsive, can only reduce the future sending rate. Packets already in transit during this interval accumulate in bottleneck queues, causing significant delay and violating the application's latency budget. For example, if bandwidth drops by 20×[1] and RTT is 50 ms, draining the backlog may take over a full second, even if the sender adapts instantly. On the other hand, purely in-network schemes like AQM are application-blind. A router lacks the application-level context. It does not know when an RTC flow is approaching its latency cliff, nor can it distinguish a high-value keyframe from a disposable delta frame. Therefore, directly dropping packets by routers is often ineffective and may even lead to a significant degradation in QoE (Quality of Experience).

To address this gap, we propose Gecko, a collaborative transport scheme that augments end-hosts with bottleneck-aware queue flushing logic. The core idea is to establish a cooperative control loop: the in-network router predicts impending congestion and alerts the sender, while the sender decides whether to command the router to flush stale packets using its application knowledge. This circumvents the tail-latency limits of pure end-to-end control and the application-blindness of pure in-network mechanisms. However, realizing such fine-grained, cross-layer cooperation raises significant practical challenges. First, how can the router alert the sender without breaking established protocols? Prior explicit coordination schemes often required router to modify packet or create out-of-band signals, compromising end-to-end integrity and hindering real-world deployment. End-to-end integrity is a crucial principle of network design, ensuring that data from the sender arrives at the receiver unaltered and confidential, without interference from intermediate nodes. Violating this principle can introduce security vulnerabilities.Second, how can a network router, which is inherently application-blind, know what delay is unacceptable for a specific flow?

We contribute two key innovations to overcome these barriers, respectively. First, Gecko redesigns the signaling channel between endpoints and routers. For the router-to-sender alert, Gecko uses a novel "feedback-echo" mechanism. When the router predicts excessive frame delay, it retransmits the last

[1]Sometimes the sudden bandwidth reduction can be up to 50× [42].

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

feedback packet (e.g., RTCP) twice. The sender interprets this temporal pattern as a congestion signal, avoiding any router-side packet modification. For the sender-to-router command, the sender embeds a flag into the RTP header, explicitly authorizing selective queue flushing. Second, to solve the application-awareness challenge, Gecko enables routers to predict frame sojourn time and adapt to each flow's latency sensitivity. The router triggers RTCP-echo alerts when the predicted delay exceeds a dynamic threshold. It then observes the sender's reaction to adapt the threshold accordingly. Over time, this feedback loop allows the router to refine its threshold per flow, aligning its behavior with the application's specific latency requirements without any explicit negotiation.

These designs make Gecko not only effective but also practical for deployment, especially in wireless edge environments such as home gateways, VR setups, and dedicated access points for cloud gaming or telemedicine. Such deployments often allow coordinated control over both endpoint and router software [19], creating an opportunity for collaborative schemes. We implement Gecko as a working prototype and evaluate it using both trace-driven simulations and real-world testbed experiments. Our results show that Gecko can reduce overall frame delay by up to 21.8%, and cut the long-tail delay frame ratio by 21%–91%, significantly improving RTC experience under volatile wireless conditions.

Our main contributions can be summarized as follows:

- We identify the performance gap in existing transport and AQM mechanisms arising from their inability to manage in-flight packets during network transitions (§II).
- We propose Gecko, a deployable end-to-network collaboration scheme that proactively flushes bottleneck queues through coordinated frame skipping decisions between endpoints and routers (§III).
- We design a lightweight, protocol-transparent signaling mechanism, including RTP packet marking for sender instructions and passive RTCP duplication for router notifications, coupled with a feedback-driven adaptive delay threshold algorithm that self-tunes without explicit configuration, avoiding explicit configuration or negotiation (§III-C, §III-D, §III-E).
- We implement Gecko and evaluate it extensively through ns-3 simulations and real-world testbeds, demonstrating robust improvements in delay consistency, tail performance, and perceived user quality (§IV).

This work does not raise any ethical issues. We follow the Menlo Report [4]. All of our experimental activities have received approval from the academic committee of our department.

II. Background and Motivation

A. Background: the Plight of RTC over Wireless Networks

RTC applications nowadays (e.g., VR and cloud gaming) demand not only low latency but also consistent low latency. In this case, even 1% of frames violating the requirement of the application could lead to severe degradation of the user's experience [28]. For these applications, the quality of experience (QoE) is directly coupled with the stability and magnitude of end-to-end delay, where even a few tens of milliseconds of unexpected jitter can degrade the user experience. However, maintaining consistently low latency is a formidable challenge in today's pervasive wireless networks like Wi-Fi and 5G [25], [9], [5].

The fluctuations in wireless networks create significant obstacles for RTC applications to have a satisfactory performance. When the extreme requirement on the consistency from RTC applications meets the fluctuating nature of wireless networks, these RTC applications are going to suffer a lot from stalls and stutters. Measurements of RTC applications reveal that wireless users encounter 2× more video rebuffering than Ethernet users [27], with performance bottlenecks often located in the wireless last mile [34]. Such a scenario is only going to be worse as an increasing number of users nowadays get access to the Internet through wireless networks.

B. Motivation: The Challenge of Reaction-Time Data Debt

The central challenge in delivering low-latency RTC over volatile wireless networks is not merely that networks are unpredictable, but how systems react to this unpredictability. We identify a key issue we term **reaction-time data debt**: the performance degradation that results from data injected during the lag between a capacity drop and the sender's congestion control reaction. The accumulated packets will be queued at the bottleneck, creating a large-scale latency surge. While numerous solutions have been proposed, they fundamentally fail because they cannot effectively manage this data debt without violating other critical system principles.

Rate adaptation has inadequacy. End-to-end congestion control algorithms, from classic TCP variants to modern RTC-focused designs like GCC [38] and Copa [3], form the first line of defense. They usually locate at the sender as shown in figure 1. Their strategy is to prevent future congestion by reducing the sending rate in response to perceived congestion signals (e.g., increasing RTT, packet loss) [3], [11], [39], [41]. However, they are inherently incapable of dealing with the debt that has already been accrued. Once packets are in the network, the sender can only wait for the bottleneck queue to slowly drain, a process that can take hundreds of milliseconds. For example, consider a typical videoconferencing scenario with a 50 ms RTT [7] and an RTC flow sending at 10 Mbps [7]. Suppose the network capacity suddenly drops to 1Mbps due to interference. Even in the best case, the congestion controller needs at least one RTT (recent efforts [27] aim to shorten this reaction time, but they cannot prevent the backlog that builds up before the rate adaptation takes effect) to detect the degradation and react to it. During this reaction period, approximately 500 kb of data (10 Mbps × 50 ms) will accumulate in the bottleneck queue, and draining this backlog alone requires at least 500 ms.

The blindness of in-network management. Active Queue Management (AQM) schemes located in the network as shown in figure 1, such as CoDel [32], operate directly on the bottleneck queue, representing a more proactive approach.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

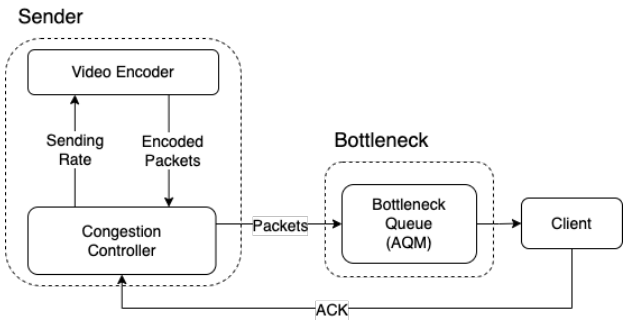


Fig. 1: The RTC video packet's network path. The congestion controller processes the congestion signals in ACK packets and adjusts the sending rate. The bottleneck-located AQM can directly control the in-flight packets.

They attempt to alleviate congestion by strategically dropping packets as packet loss signals [32], [16], [14]. However, the fundamental limitation of AQM is its blindness. It does not know when an RTC flow is approaching its latency cliff, nor can it distinguish a high-value keyframe from a disposable delta frame. Consequently, its interventions are untargeted and often suboptimal for RTC, sometimes even exacerbating QoE degradation by dropping critical data. There is also a line of work on adaptive buffer sizing [44], which will try to adapt the size of the bottleneck queue based on network conditions. However, they are more designed to *absorb* the fluctuations in the network than *clear* the stuck packets, which will still confront the same issue.

Explicit coordination faces deployment dilemma. A third category of solutions attempts to bridge this gap through explicit end-to-network coordination, as seen in datacenter protocols like DCQCN [46] and HPCC [26]. These systems create direct communication channels (e.g., using ECN marking or custom headers) to allow the network to provide precise feedback to the sender. However, those approaches face a critical deployment dilemma in the public Internet, like XCP [23] and RCP [12]. The internet's success is built on a simple core with intelligence at the edges. By altering the data packets themselves, they violate the principle of end-to-end integrity. This violation creates a brittle dependency. The signal faces the risk of loss if even a single router on the path misunderstands the custom marking.

Our analysis leads to a clear set of requirements for a viable solution. It must be: (1) effective, with the ability to actively clear accumulated data debt; (2) application-aware, leveraging end-host intelligence to guide its actions; and (3) pragmatically deployable. The third requirement implies that the solution should be incrementally deployable at network edges and must preserve end-to-end packet integrity, allowing in-network devices to act on behalf of a flow, but never to alter its packets. These principles motivate a new paradigm for a system that enables application-aware, in-network intervention through a protocol-transparent signaling channel that requires no packet mutation.

III. Methodology

A. Design Challenges and Our Key Ideas

Our goal is to design a practical, application-aware in-network intervention system. This requires addressing two key challenges.

Challenge 1: Building a practical signaling mechanism between the sender and the router. The sender-to-router communication is straightforward as the sender can embed information into the headers of outgoing packets (e.g., RTP or TCP, which does not encrypt), which the router can read. The critical part is the reverse path: how can the router promptly signal an alert to the sender? Embedding a control signal into a regular feedback packet, like RTCP or TCP ACK, is problematic. It may require modifying the packet, and the router must wait for the next feedback packet to arrive, introducing unacceptable delays. Conversely, generating a new, separate control packet would introduce the costs and risks of out-of-band signaling, potentially triggering firewalls and complicating protocol design. To overcome this, Gecko introduces a lightweight and protocol-agnostic **feedback-echo** mechanism. The router only needs to cache the most recent feedback packet it forwards from the receiver to the sen der. Upon triggering an alert, it immediately re-transmits this cached packet to the sender twice in rapid succession. This approach is universally applicable to any feedback protocol because it treats the packet as an opaque object, and we use RTCP-echo in our implementation. It guarantees that a signal can be sent immediately without packet construction overhead, and since the echoed packet is identical to a legitimate feedback packet, it fully preserves end-to-end integrity and avoids any out-of-band signaling issues. The sender, in turn, is designed to interpret the immediate arrival of duplicate feedback packets as a network congestion alert.

Challenge 2: Making robust and efficient congestion alerts. This involves two critical questions: how to predict congestion, and what level of predicted congestion should trigger an alert. First, the delivery rate in wireless networks is notoriously volatile [27], [21], making per-packet delay predictions unreliable and computationally expensive. To address this, we shift to a more stable and efficient **frame-level sojourn time prediction**. The sender marks frame boundaries, allowing the router to predict the total time an entire frame will spend in its queue. This metric is inherently more robust to network jitter and directly aligns with the application's experience, as a frame is typically only useful once fully received. Second, and more importantly, a one-size-fits-all trigger threshold is impractical. Different RTC applications have vastly different latency tolerances. Therefore, the router should not only predict delay but also learn the specific latency sensitivity of each application flow. To achieve this, we introduce a dynamic threshold adaptation mechanism. The router observes the sender's responses to its alerts and interprets this feedback to continuously adjust the trigger threshold for that flow. This creates a closed-loop system where the router's intervention policy automatically

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

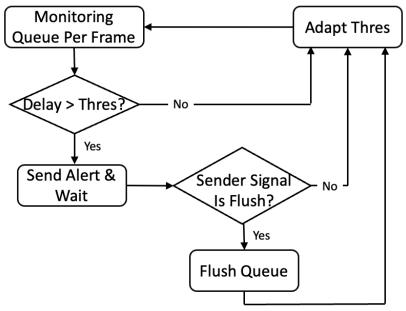


Fig. 2: Gecko's queue flushing logic.

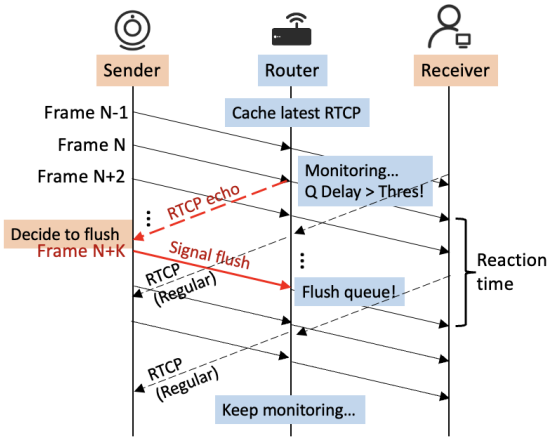


Fig. 3: The Gecko workflow. The router initiates a signal via RTCP-echo when predicted queue delay exceeds a threshold. The sender responds with a flush command embedded in a data packet, prompting the router to clear its queue for the RTC flow.

adapts to become application-aware without requiring explicit negotiation.

B. System Overview

In this section, we present Gecko, a new transport scheme for application-aware in-network queue flushing logic. Gecko materializes the design philosophy of enabling sender-side decisions with router-side execution, creating a collaborative framework that effectively and safely eliminates RTC data debt, while preserving end-to-end integrity and ensuring incremental deployability.

As depicted in Figure 2 and 3, the Gecko coordination framework allows the application, which possesses the ultimate knowledge of its own performance requirements and content characteristics, to request an immediate and precise queue-clearing action from the network. The Gecko-enabled router, typically the last-hop wireless router, acts as a capable agent that executes this request without altering the data packets themselves, thus respecting the end-to-end principle. The workflow operates in a precise, three-phase cycle:

1) **In-network latency prediction and signaling:** the router continuously monitors the queue of active RTC flows and predicts the sojourn time of newly arrived frames. When this predicted sojourn time exceeds a dynamically adapted threshold, the router signals impending congestion by sending a pair of duplicate RTCP feedback packets to the sender. This "RTCP echo" acts as a protocol-transparent alert without altering the normal control logic.
2) **Sender-side decision:** upon receiving this alert, the sender evaluates its content and performance requirements to decide whether skipping queued frames is preferable to waiting. It then communicates its decision by marking subsequent RTP packets with minimal in-band flags.
3) **Router execution and adaptation:** the router interprets these flags as explicit flush commands for the corresponding flow and immediately clears stale packets to eliminate accumulated delay. In addition, the router adjusts its latency threshold over time based on the sender's responses, aligning its behavior with the application's actual sensitivity without requiring explicit negotiation.

C. The Gecko Signaling Mechanism: Deployable End-to-Network Coordination

The effectiveness and deployability of Gecko hinge on its novel signaling mechanism. Unlike traditional cross-layer designs that rely on explicit, often complex, negotiation protocols, Gecko establishes a lightweight, bidirectional coordination loop. This loop allows the router to provide a real-time congestion alert and the sender to issue a precise, in-band command in response. The entire mechanism is designed to be transparent to legacy network devices and to preserve the end-to-end integrity of the RTC stream, ensuring it can be incrementally deployed in real-world networks.

Router-to-sender alert signal. The first phase of the Gecko control loop is the alert signal, initiated by the network itself.

1) Latency prediction and trigger. A Gecko-enabled router monitors the queue for identified RTC flow. It predicts the total sojourn time of each arriving frame and triggers a congestion alert when the predicted delay exceeds a dynamically adapted threshold (detailed in §III-D).
2) The duplicate RTCP-echo signal. To communicate this alert to the sender without modifying any packets or introducing a new protocol, Gecko employs a novel, timing-based signaling technique. The router maintains a cache of the most recent RTCP feedback packet it has forwarded from the receiver to the sender. Upon triggering an alert, the router immediately re-transmits (or "echoes") this cached RTCP packet to the sender twice in rapid succession, as shown in Figure 3.

Sender-to-router embedded commands. Upon receiving the alert, the decision-making authority resides with the sender, which has the full application context.

1) Signal interpretation and decision. The Gecko-enabled sender is programmed to recognize the arrival of two identical RTCP packets within a very short time window as a congestion alert from the network. Once the alert is confirmed, the application can make a decision. For a video conferencing application, this could involve checking if the frames currently in the network queue are disposable (P- or B-frames) and if flushing them would be less detrimental to user experience than enduring the

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

high latency. The details of such a demo RTC application are further depicted in §III-E.

2) In-band command flag. Whether the sender decides to proceed with a flush, it will communicate its decision back to the router. To do this, it utilizes a pre-defined flag in the header of the next outgoing RTP packet. This can be implemented using 2 bits in the RTP header extension mechanism, which is designed for such custom signaling and is ignored by legacy systems.

In-network execution. The final phase of the loop is the execution of the flush command at the router, which must be both precise and safe.

1) Command detection and execution. After sending an alert signal, the Gecko router enters a brief waiting period to wait for the sender command. During this period, it continues to monitor the queue, but no longer sends repeated alerts. Instead, it inspects the headers of incoming RTP packets from the corresponding sender. If it detects the "flush" flag, it immediately performs an atomic flush operation on the queue for that specific flow. This involves dropping all packets currently buffered for that flow. If the sensitive period expires without receiving a flush command, the router assumes the sender has opted to endure the delay, and it returns to its normal monitoring state.
2) Safety and integrity. The flush operation is inherently safe for two reasons. First, it is highly selective, targeting only the queue of the flow that participated in the signaling handshake, leaving other flows on the router unaffected. Second, and most critically, the router only drops packets and never mutates their content. This preserves the end-to-end integrity of the transport protocol, as the receiver will simply see a gap in the packet sequence, which is a normal occurrence in packet networks that RTC applications are already built to handle.

By closing this loop, Gecko achieves a state of informed and rapid intervention. The router provides the timely trigger, the sender provides the application-aware intelligence, and the router performs the powerful execution, resolving the coordination dilemma.

D. Router Task: Congestion Prediction and Adaptive Triggering

To build a robust and efficient alert mechanism, a router must answer the two questions: (1) how to accurately predict congestion, and (2) what level of predicted congestion should trigger an alert.

Frame sojourn time prediction. A naive approach is to predict the queuing delay for each individual packet. However, per-packet predictions are not only computationally expensive for the router but also unreliable, as they are susceptible to transient network jitter, especially in wireless networks [27], [21]. Moreover, they do not align with the application's perception of latency. For most RTC applications, particularly video streaming, latency is experienced at the frame level because a frame is only decodable after its last packet has been received. Therefore, the true experienced latency is the sojourn time of the entire frame, i.e., the duration from the first packet's arrival at the queue to the last packet's departure. This metric naturally smooths out the microtimescale jitter and directly reflects the application's experience.

The sender marks frame boundaries using the marker bit in the RTP header. The router can use this bit to identify the start of a new frame. The router predicts the sojourn time T_S for each frame, which is composed of two parts: the time to drain the existing queue ($T_{\text{drain}} = \frac{Q_{\text{depth}}}{R_{\text{out}}}$) and the time to transmit the frame itself ($T_{\text{tran}} = \frac{S_{\text{frame}}}{R_{\text{out}}}$).

Dynamic threshold adaptation. Once the router has a prediction of the frame sojourn time, it must decide if this predicted time warrants an alert. Traditional in-network management schemes, such as CoDel AQM [32], often rely on static or manually configured queue thresholds. This one-size-fits-all approach is flawed in RTC scenarios. For example, 100 ms might be too high for a fast-paced cloud gaming application, leading to unacceptable lag, while being unnecessarily aggressive for a more buffered video stream, causing needless packet loss. Gecko employs a dynamic threshold adaptation mechanism to learn the application's specific latency tolerance. This mechanism treats the sender's response to an alert as implicit feedback. This process follows an AIMD-inspired control loop, a well-established approach known for its stability and responsiveness in network environments [15].

1) Initialization. Each monitored flow f is initialized with a default latency threshold at 100 ms. Two counters are updated for each flow: *acks*, the number of times the sender responded with a flush command, and *nacks*, the number of times the sender responded with a no-flush command or timeout.
2) The router calculates the flush ratio $r = \frac{\text{acks}}{\text{acks+nacks}}$ within a sliding window of the most recent 5 alerts. It then adjusts the threshold T_f based on both the current value of r and its recent trend. If the flush ratio is high and rising ($r > 0.5$ and $r > r_{\text{old}}$), it suggests the application is becoming more sensitive to latency. The router then multiplicatively decreases the threshold: $T_f = \max(\beta T_f, T_{\min})$. If the flush ratio is low and decreasing ($r < 0.5$ and $r < r_{old}$), it suggests the application is becoming more tolerant of delay. The router additively increases its threshold: $T_f = \min(\Delta T + T_f, T_{\max})$. In all other cases, like stable ratio, or ratio and trend are contradictory, the threshold remains unchanged to avoid oscillation.[2]

This self-adaptation allows Gecko to react more intelligently, achieving application awareness without requiring explicit negotiation, making it practical for real-world deployment.

E. Sender Task: Illustrative End-Host Implementaton

In this section, we present an illustrative implementation of an RTC application. The core task of the sender is to map these

[2]Parameters: we use $\beta = 0.7$, a commonly adopted value in AIMD-based congestion control schemes. ΔT is a fixed additive step size set to 10 ms.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

decisions to their resulting impact on application performance and choose the action that yields a better outcome. We demonstrate a simple yet effective decision logic to illustrate this principle. In detail, once an RTC session is established and has passed the startup phase, the sender enters a stable state. At this time, the reception of a Gecko alert signifies that the queuing delay at the bottleneck is about to exceed a limit. Upon identifying an alert, the end-host application must decide between two actions: sending a *no-flush* signal to tolerate the delay, or a *flush* signal to clear the queue.

Our illustrative implementation employs an online A/B testing strategy. When the first alert is received, the sender defaults to sending a *no flush* command. It then measures the performance after the decision, specifically counting the number of frames whose playback delay exceeds the application-specific tolerance (we use 200ms). We denote this count as N_{NF_bad}. Upon receiving the next alert, the sender issues a *flush* command (unless an I-frame is at risk). It again measures the performance, but this time N_{F_bad} includes both the frames that are skipped due to the flush command and subsequent frames that still exceed the delay tolerance. After these two cycles, the sender maintains the time-decayed moving averages $\bar{N}_{F\_bad}$ and $\bar{N}_{NF\_bad}$. If the current decision was *flush* and the resulting N_{F_bad} is greater than $\bar{N}_{NF\_bad}$, it implies that flushing is currently less beneficial than tolerating the delay. The next decision will be *no-flush*. Conversely, if the current decision was *no-flush* and N_{NF_bad} is greater than $\bar{N}_{F\_bad}$, it indicates that flushing would have been a better choice. The next decision will be *flush*. To ensure the system continues to explore and adapt to changing conditions, if the same command is issued for five consecutive alerts, the sender will forcibly issue the opposite command to refresh its performance measurements of the alternative strategy.

This example illustrates the role of the end-host in the Gecko architecture. More sophisticated mechanisms could be built in practice, incorporating other network statistics or user-defined metrics, but the core principle of mapping actions to performance outcomes remains the same.

Feasibility concerns. Gecko is able to keep consistent with the network and has low consumption. Gecko-router does not make any assumptions about the sending policy, network topology, and incoming traffic. It is designed to be resilient to other flows and devices in the network. The router performs predictions only on identified RTC flows. For other types of flows, Gecko's bottleneck program does not maintain the enqueue and dequeue states, requiring few resources. For RTC streams, Gecko restricts the prediction step to once per frame. At common frame rates of 30 or 60 fps, the prediction frequency is a modest load. Our experiments in §IV-E prove that the overhead is acceptable for the bottleneck device.

IV. Evaluation

We evaluate Gecko with both ns-3 simulations and testbed experiments. Ns-3 simulations are employed to finish a proof-of-concept, and then our testbed experiments and user study further confirm the improvement Gecko brings.

We first introduce our implementation and experiment settings in §IV-A. Then we analyze the experiment results to answer the following questions:

1) *Can Gecko significantly reduce the frame delay by draining the queue fast?* Through our experiment, we aim to find out the actual benefits of draining the queue at the bottleneck. (§IV-B)
2) *Can Gecko improve the overall performance and tail performance with various real-world wireless traces?* To answer this question, we evaluate the performance of Gecko via a series of trace-driven experiments. We find that Gecko can reduce the overall delay by 21.8% and the tail-delay frame ratio by 25% to 91%. (§IV-C)
3) *Can Gecko gain good performance in real world?* We conducted real-world testbed experiments. We adopt several metrics to estimate the properties of Gecko and compare Gecko with other schemes. (§IV-D)
4) *Is Gecko robust in terms of overhead and fairness?* We find that Gecko has acceptable overhead and can protect the fairness between flows. (§IV-E)

A. Experimental Setup

Simulation setup. The implementation consists of a bottleneck queuing program deployed on edge routers and application-level decision logic on the sender. We implement a WebRTC-like application on ns-3, including video sender and receiver components that communicate via RTP/RTCP.

Testbed setup. Our testbed consists of three machines: an OpenWrt router running the Gecko queuing discipline, a sender running the custom WebRTC application, and a receiver. For the network-side implementation, we develop a queuing discipline as an OpenWrt extension package using the opkg package manager [1], enabling deployment on commodity wireless routers. We use the native WebRTC library in Google Chrome, leveraging the insertableStreams API (available in Chromium version 111.0.+) to support frame skipping triggered by the flush command. The application supports dominant codecs, including H.264/265 and VP8/9. Network conditions are emulated using tc with 50ms RTT and 10Mbps bandwidth. Video streams operate at 30 fps. We evaluate tail performance using frame delay and SSIM distributions, as well as system overhead through CPU and memory usage.

Baselines. We compare Gecko against three approaches: PfifoFast (default FIFO queuing), CoDel (AQM with 5ms target delay), and Zhuge [27], a recent in-network optimization scheme for RTC. These baselines represent different paradigms: passive queuing, active queue management, and application-aware optimization.

Congestion Control. To demonstrate Gecko's generality, we evaluate it with two representative RTC congestion control algorithms: GCC [6](Google Congestion Control, the WebRTC default) and NADA [45] (Network-Assisted Dynamic Adaptation, a delay-oriented CCA).

Network Traces. We use six real-world wireless traces: two WiFi (W1, W2) and four cellular (C1-C4). The C3 traces are

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

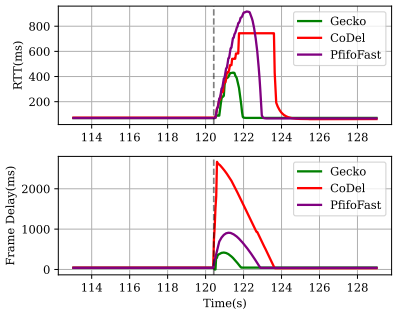


Fig. 4: We cut the bandwidth at 120.43 s. The figure presents the packet RTT and frame delay change during the period. Among them, Gecko reacts the fastest to the bandwidth change.

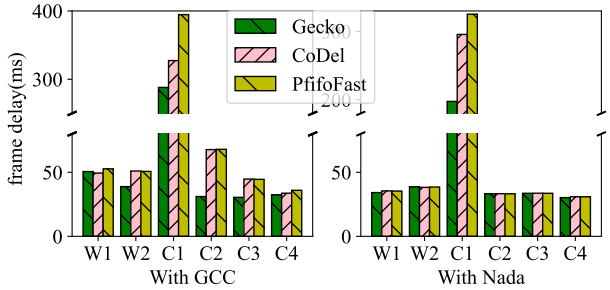


Fig. 5: The average frame delay results of the trace-driven simulations with GCC and Nada congestion controller respectively.

from Mahimahi [30], while others are collected under diverse real-world conditions [27]. Each trace contains timestamped bandwidth and delay measurements.

B. Micro-benchmark: The Advantage of Gecko

We evaluate the benefit of fast queue flushing using a 200s synthetic trace where the bandwidth drops sharply from 60Mbps to 6 Mbps at 120.43s (Fig. 4). All schemes use a 1000-packet bottleneck buffer and GCC as the CCA. We measure RTT and frame delay.

Gecko reacts almost immediately to the bandwidth change, restoring both RTT and frame delay within 1.6s, while PfifoFast and CoDel recover much slower (2.5s and 3.8s, respectively) due to gradual queue draining. This confirms that Gecko can promptly detect network changes and aggressively drain the queue, achieving the fastest return to low-latency operation. Frame delay exhibits the same trend, as Gecko proactively drops at the frame level. Note that CoDel performs the worst because GCC is delay-oriented and already reacts to RTT inflation. Additional random drops in CoDel introduce unnecessary retransmissions and disrupt frame integrity, forcing incomplete frames to wait for recovery and further increasing frame delay.

C. Trace-driven Simulation

For the trace-based simulation, the results are shown in Figure 5, 6, 7 and 8.

Overall performance: Gecko outperforms all the baselines for all the traces with regard to the sending rate, average frame delay, and tail frame delay. Generally, Gecko with GCC can reduce the overall frame delay by 21.8% on average. It can

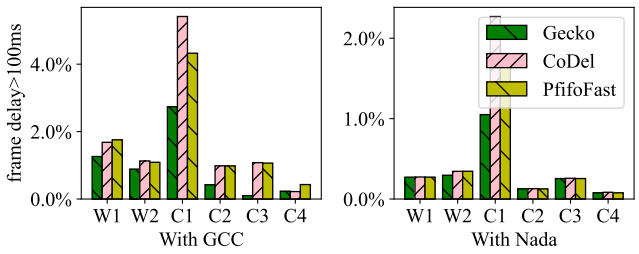
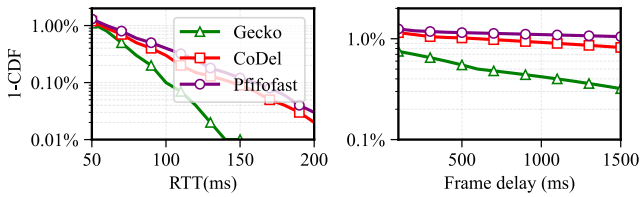
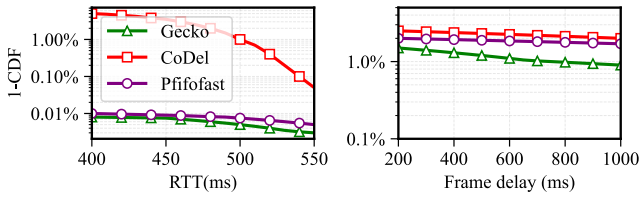


Fig. 6: The percentage of long-delay frames in the trace-driven simulations with GCC and Nada.



(a) The 1-CDF of tail RTT and frame delay with W2 trace.



(b) The 1-CDF of tail RTT and frame delay with C1 trace.

Fig. 7: The 1-CDF of **tail** RTT and frame delay in the simulation with two groups of traces.

reduce the ratio of long-delay frames by 25% to 91% for different groups of traces. Also, Gecko can reduce the tail frame delay and the tail RTT distribution significantly.

Average frame delay: Figure 5 shows the average frame delay results of the trace-driven tests. In most of the cases, Gecko achieves the best average frame delay among the three schemes. In group W1, the average frame delay of CoDel is slightly better than Gecko by about 2%. That is probably because the sending rate of Gecko is higher than CoDel (according to Figure 8) by about 8%. The most outstanding improvement is in the GCC test of group C2 that Gecko nearly cuts in half the average frame delay of CoDel and PfifoFast.

Long-delay frame ratio: Specifically, we count the ratio of long-delay frames whose frame delays are more than 100ms. Gecko significantly reduces the long-delay frame percentage by up to 91% in the GCC test of group C3. Gecko shows the capability of reducing the tail frames under all of the groups of network traces.

As for Nada, both the frame delay and the tail frame ratio results of the three schemes are nearly equal in most of the groups. We notice that Nada's frame delay and tail delay are both better than GCC, but the sending rate is rather low. Our preliminary guess is that Nada is too sensitive to link

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

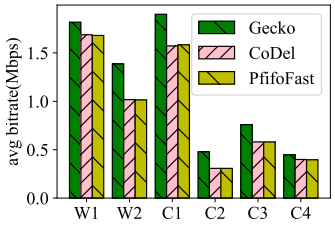


Fig. 8: The overall average bitrate of the trace-driven simulations. Gecko has an evident advantage over other queue disciplines.

fluctuations. In our experiments, the Nada controller keeps a low link capacity, giving the queue disciplines little chance to work. Only under poor network conditions, e.g., in group C1, will the queue accumulate. In that case, Gecko also outperforms other schemes.

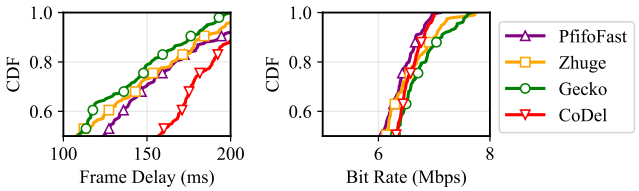
Tail performance: We calculate the 1-CDF of the tail RTT and frame delay and present the results of two groups among six in Figure 7. The tail distributions of Gecko's RTT and frame delay are both better than those of CoDel and PfifoFast. The highest delay in the figure is 1000ms because our application sets 1000ms as the delay deadline in the trace simulations. Usually, a delay of 1000ms in RTC application is unacceptable. Lost frames and frames with over 1000ms delay are regarded as the same when we count the tail distribution in figure 7.

Bitrate: Our simulation result in Figure 8 shows that Gecko can help the flow obtain more bitrate than other queue disciplines by 8% to 67% among all the groups of experiments. Gecko bottleneck can notify the sender about the congestion. When congestion occurs, the Gecko control loop is shortened by a queuing delay time. Therefore, with the same CCA, flows with Gecko can recover fastest from congestion and consequently achieve more link capacity.

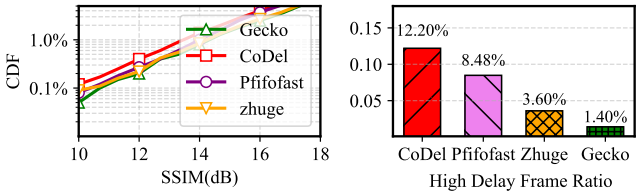
D. Real-world Performance

We evaluate the end-to-end performance of Gecko, Zhuge, PfifoFast, and CoDel using five distinct video transmitted over a real-world wifi router. The results are summarized in Figure 9.

Figure 9(a) presents the CDFs of frame delay and bitrate. Among all schemes, Gecko consistently achieves the lowest frame delay and the highest, most stable bitrate. The majority of frames under Gecko experience delays below 150 ms, indicating superior latency performance compared to the baselines. Furthermore, Gecko maintains a higher and more consistent bitrate, demonstrating robust adaptability to bandwidth fluctuations. Figure 9(b) further examines tail performance. Gecko reduces the proportion of high-delay frames (delay>200 ms) to 1.4%, representing a 61% reduction compared to the best baseline. In contrast, CoDel and PfifoFast exhibit significantly higher high-delay frame ratios (12.2% and 8.5%, respectively). In terms of video quality, Gecko also delivers clear advantages. The tail SSIM CDF reveals that Gecko significantly reduces the occurrence of low-quality frames (SSIM<10dB), achieving a 66% reduction compared to the best baseline.



(a) The 1-CDF of frame delay and bitrate in real-world experiments.



(b) The 1-CDF of tail SSIM and ratio of high-delay frames.

Fig. 9: The results of real-world experiments.

Scheme	CPU	Memory(MB)
Gecko	2%	3.1
Zhuge	2%	5.3
CoDel	1%	2.8
Fifo	1%	1.3

TABLE I: Resource overhead during RTC video delivery.

E. Robustness Analysis

To evaluate the resource overhead introduced by Gecko, we conduct a set of experiments under the same RTC video delivery scenario described in Section IV-D. Table I summarizes the CPU and memory usage of each scheme. Fifo serves as the baseline, representing minimal overhead. Compared to Fifo, Gecko incurs slightly higher resource usage, but remains comparable to CoDel and lower than Zhuge. Specifically, Gecko uses 2% CPU and 3.1MB of memory, indicating that its proactive mechanisms do not impose excessive computational or memory demands. These results demonstrate that Gecko achieves low-latency performance with modest system overhead, making it practical for real-world deployment.

V. Discussion

Generality and practicality. One practicality concern is that modifying the sender and the bottleneck together is difficult for the service provider. We note that the structure of wireless networks makes it possible to locate the bottleneck, usually at the wireless hop. One possible method to extensively deploy our scheme in the real world is for RTC service providers to cooperate with AP providers. We also recommend the AP providers preemptively add the scheme as an extra feature cause it does not conflict with other properties in the AP, but can provide more efficient transport for specific RTC traffic. Our method suggests a point of view that when end-to-end schemes are limited, an attempt to share limited states and control signals between the network and application could make sense.

Transport protocols. Gecko is designed for dominant transport protocols in RTC service based on TCP and UDP. Our

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

implementation is based on RTP/RTCP. Actually, it can adapt to other protocols as long as it can locate the packet header bits. However, newly emerging protocols put forward further challenges for the scheme. For example, QUIC encrypts the packet header as well as the payload to guarantee end-to-end security. Since this feature prevents the in-network device from sniffing the packet header, dropping by frame is quite difficult for QUIC flows. We consider strict end-to-end transport and service-customized efficient framework (e.g., Gecko) as two parallel trends of future network development.

Security issues. While coupling the bottleneck with the endpoint introduces potential security concerns, our design is explicitly crafted to preserve end-to-end integrity. Nevertheless, the threat of a malicious man-in-the-middle (MitM) attacker remains a concern for both endpoints. Although our approach avoids direct packet modification, thereby posing less risk compared to schemes that alter traffic, its security implications warrant further exploration. A comprehensive analysis of the collaborative scheme's resilience against various attack vectors is expected in future work.

Fairness. We implement Gecko based on fair queue. Fairness is naturally ensured since the flows are hashed into separate queues. To prove this, we add a new Iperf flow to share the bottleneck with an Gecko flow. The steady average bitrate decreased by 63.1%, which is comparatively fair. [3]

VI. Related Work

Wireless network optimization. Wireless transport is a long-time-discussed topic. Previous wireless optimization schemes like [40], [33] emphasize the special physical link characteristics and design customized CCAs to optimize the end-to-end performance. Several innovations also focus on video performance in wireless networks. [37], [24] investigate the video codec features in wireless networks. [18] further modifies the video codec to change the video bitrate to adapt to the network conditions dynamically. Compared to them, Gecko provides an earlier and more precise reaction to wireless network congestion with in-network support.

RTC optimization. Plenty of innovations have focused on utilizing the performance of RTC service. Proposals in [19], [35] target the speed of bitrate streaming at the encoder. [8], [17] focus on error control and aim to optimize the FEC policy in RTC transport. [13] proposes a solution developed on the routing and data plane behavior to reduce packet loss in video conferencing. None of these works aim to provide in-network RTC transport enhancement. In contrast, Gecko emphasizes the queuing delay, which is the most immediate consequence of the congestion. Through fast draining the queue, we make an effort to react fast to network congestion and improve the RTC performance, especially tail performance.

Low-latency transport protocols. Low latency as well as high throughput is an essential goal in transport design. One major cause of latency deterioration in the network queue is bufferbloat [20]. Bufferbloat is the phenomenon of packets filling up the buffer resulting in a high queuing delay. In order to mitigate bufferbloat and avoid the unpleasant queuing delay, plenty of delay-oriented CCAs have been designed. Previous schemes like Sprout [39], Verus [43] and Copa [3] tend to observe the packet-level delay change to discover the congestion. Gecko does not focus on the CCA, but it can work with most of the existing end-to-end CCA schemes. Moreover, we suppose that the Gecko notification can act as a congestion signal to help with the CCA. We leave this tight coordination between Gecko and CCA for future work.

In-network assistance. A variety of in-network schemes including AQMs have been proposed to reduce queuing delay directly. Classic AQM methods like CoDel [32] and RED [16] tend to discover bufferbloat at the bottleneck and drop queuing packets in advance. However, they fail to manage the packets on the demands of the flows. Diffserv [22] aims to classify and manage different network traffic. It gives some traffic priority according to their Class of Service. Gecko does not aim to prioritize flows. Instead, it focuses on mitigating the bufferbloat and enhancing the flow-level performance.

VII. Conclusion

In this paper we present Gecko, an application-network collaboration scheme for RTC services in wireless network. Gecko achieves extremely low latency through three main steps: (1) the bottleneck creates congestion notifications, (2) the sender makes decisions, and (3) the bottleneck fast flush the queue. This mechanism can go beyond the traditional approaches because it can both directly operate at the bottleneck queue and comply with the application requirements. In our trace simulation, Gecko is able to achieve low frame delay and high throughput for various wireless traces. In our testbed experiments, it can actually optimize the real-world video delay and quality as well. Particularly, Gecko can achieve low tail latency both in the simulation and the testbed evaluation by 25% to 91% in different scenarios.

Acknowledgments

This work was supported by the National Key R&D Program of China (Grant 2025YFE0201000), the National Natural Science Foundation of China (Grant 62221003 and Grant 62502471), the Shandong Provincial Natural Science Foundation, China (Grant ZR2024LZH011), the Research Project of Provincial Laboratory of Shandong, China (Grant SYS202201), and the Guangdong NSF Project (Grant 2025A1515010460). Mingwei Xu and Enhuan Dong are the corresponding authors of the paper.

References

[1] Opkg package manager. https://openwrt.org/docs/guide-user/additional-software/opkg, 2023.

[2] R. Adams. Active Queue Management: A Survey. *IEEE Communications Surveys & Tutorials*, 15(3):1425–1476, 2012.

[3] V. Arun and H. Balakrishnan. Copa: Practical Delay-Based Congestion Control for the Internet. In *USENIX NSDI*, pages 329–342, 2018.

[3]The unfairness can be resulted from many factors, such as CCA, queuing discipline and flow joining time. Fair queue cannot guarantee that the flows share absolutely fair bandwidth, but it can mostly prevent them from starvation.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.

[4] Michael Bailey, Aaron Burstein, KC Claffy, and et al. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. *Homeland Security: Science and Technology*, 2012.

[5] A. Bhartia, B. Chen, F. Wang, D. Pallas, R. Musaloiu-E, T. Lai, and H. Ma. Measurement-based, Practical Techniques to Improve 802.11 ac Performance. In *ACM IMC*, pages 205–219, 2017.

[6] G. Carlucci, L. Cicco, S. Holmer, and S. Mascolo. Congestion Control for Web Real-time Communication. *IEEE/ACM ToN*, 25(5):2629–2642, 2017.

[7] H. Chang, M. Varvello, F. Hao, and S. Mukherjee. Can You See Me Now? A Measurement Study of Zoom, Webex, and Meet. In *ACM IMC*, pages 216–228, 2021.

[8] K. Chen, H. Wang, S. Fang, X. Li, M. Ye, and H. Chao. RL-AFEC: Adaptive Forward Error Correction for Real-time Video Communication based on Reinforcement Learning. In *ACM MMSys*, pages 96–108, 2022.

[9] Y. Daldoul, D. Meddour, and A. Ksentini. Performance Evaluation of OFDMA and MU-MIMO in 802.11 ax Networks. *Computer Networks*, 182:107477, 2020.

[10] Sandesh Dhawaskar Sathyanarayana, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Converge: Qoe-driven multipath video conferencing over webrtc. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 637–653, 2023.

[11] M. Dong, Q. Li, D. Zarchy, P. Godfrey, and M. Schapira. PCC: Re-architecting Congestion Control for Consistent High Performance. In *USENIX NSDI*, pages 395–408, 2015.

[12] Nandita Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer, 2008.

[13] A. Elmokashfi, E. Myakotnykh, J. Evang, A. Kvalbein, and T. Cicic. Geography Matters: Building an Efficient Transport Network for a Better Video Conferencing Experience. In *ACM CoNEXT*, pages 369–380, 2013.

[14] Wu-chang Feng, Kang G Shin, Dilip D Kandlur, and Debanjan Saha. The blue active queue management algorithms. *IEEE/ACM transactions on networking*, 10(4):513–528, 2002.

[15] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. Technical report, 2004.

[16] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM ToN*, 1(4):397–413, 1993.

[17] S. Fong, S. Emara, B. Li, A. Khisti, W. Tan, X. Zhu, and J. Apostolopoulos. Low-latency Network-adaptive Error Control for Interactive Streaming. In *ACM MM*, pages 438–446, 2019.

[18] S. Fouladi, J. Emmons, E. Orbay, C. Wu, R. Wahby, and K. Winstein. Salsify: Low-latency Network Video Through Tighter Integration between a Video Codec and a Transport Protocol. In *USENIX NSDI*, pages 267–282, 2018.

[19] S. Fouladi, R. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *USENIX NSDI*, volume 17, pages 363–376, 2017.

[20] J. Gettys. Bufferbloat: Dark Buffers in the Internet. *IEEE Internet Computing*, 15(3):96–96, 2011.

[21] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. {ABC}: A simple explicit congestion controller for wireless networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 353–372, 2020.

[22] Dan Grossman. New Terminology and Clarifications for Diffserv. Technical report, 2002.

[23] Dina Katabi. *Decoupling congestion control and bandwidth allocation policy with application to high bandwidth-delay product networks*. PhD thesis, Massachusetts Institute of Technology, 2003.

[24] I. Kofler, M. Prangl, R. Kuschnig, and H. Hellwagner. An H. 264/SVC-based Adaptation Proxy on a WiFi Router. In *NOSSDAV*, pages 63–68, 2008.

[25] K. Lee. Performance Analysis of the IEEE 802.11 ax MAC Protocol for Heterogeneous Wi-Fi Networks in Non-saturated Conditions. *Sensors*, 19(7):1540, 2019.

[26] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proc. ACM SIGCOMM*, pages 44–58, 2019.

[27] Z. Meng, Y. Guo, C. Sun, B. Wang, J. Sherry, H. Liu, and M. Xu. Achieving Consistent Low Latency for Wireless Real-time Communications with the Shortest Control Loop. In *ACM SIGCOMM*, pages 193–206, 2022.

[28] Zili Meng, Tingfeng Wang, Yixin Shen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Enabling high quality real-time communications with adaptive frame-rate. In *USENIX NSDI*, 2023.

[29] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. A variegated look at 5g in the wild: performance, power, and qoe implications. In *Proc. ACM SIGCOMM*, pages 610–625, 2021.

[30] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*, pages 417–429, 2015.

[31] Yunzhe Ni, Zhilong Zheng, Xianshang Lin, Fengyu Gao, Xuan Zeng, Yirui Liu, Tao Xu, Hua Wang, Zhidong Zhang, Senlang Du, et al. Cellfusion: Multipath vehicle-to-cloud video streaming with network coding in the wild. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 668–683, 2023.

[32] K. Nichols and V. Jacobson. Controlling Queue Delay. *Communications of the ACM*, 55(7):42–50, 2012.

[33] S. Park, J. Lee, J. Kim, J. Lee, S. Ha, and K. Lee. ExLL: An Extremely Low-latency Congestion Control for Mobile Cellular Networks. In *ACM CoNEXT*, pages 307–319, 2018.

[34] C. Pei, Y. Zhao, G. Chen, R. Tang, Y. Meng, M. Ma, K. Ling, and D. Pei. WiFi Can be the Weakest Link of Round Trip Network Latency in the Wild. In *IEEE INFOCOM*, pages 1–9, 2016.

[35] Y. Sambe, S. Watanabe, D. Yu, T. Nakamura, and N. Wakamiya. High-speed Distributed Video Transcoding for Multiple Rates and Formats. *IEICE Transactions on Information and Systems*, 88(8):1923–1931, 2005.

[36] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[37] T. Stockhammer and M. Hannuksela. H. 264/AVC Video for Wireless Transmission. *IEEE Wireless Communications*, 12(4):6–13, 2005.

[38] George Suciu, Stefan Stefanescu, Cristian Beceanu, and Marian Ceaparu. Webrtc role in real-time communication and video conferencing. In *IEEE Global Internet of Things Summit (GIoTS)*, 2020.

[39] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *USENIX NSDI*, 2013.

[40] Y. Xie, F. Yi, and K. Jamieson. PBE-CC: Congestion Control via Endpoint-centric, Physical-layer Bandwidth Measurements. In *ACM SIGCOMM*, pages 451–464, 2020.

[41] Yaxiong Xie, Fan Yi, and Kyle Jamieson. Pbe-cc: Congestion control via endpoint-centric, physical-layer bandwidth measurements. In *Proc. ACM SIGCOMM*, pages 451–464, 2020.

[42] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption. In *Proc. ACM SIGCOMM*, pages 479–494, 2020.

[43] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, and C. Görg. Adaptive Congestion Control for Unpredictable Cellular Networks. In *ACM SIGCOMM*, pages 509–522, 2015.

[44] Yueping Zhang and Dmitri Loguinov. Abs: Adaptive buffer sizing for heterogeneous networks. In *Proc. IEEE IWQoS*, pages 90–99, 2008.

[45] X. Zhu, R. Pan, M. Ramalho, and S. Mena. Network-assisted Dynamic Adaptation (NADA): a Unified Congestion Control Scheme for Real-time Media. *RFC8698*, 2020.

[46] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proc. ACM SIGCOMM*, pages 523–536, 2015.

[47] Xutong Zuo, Yong Cui, Xin Wang, and Jiayu Yang. Deadline-aware multipath transmission for streaming blocks. In *Proc. IEEE INFOCOM*, pages 2178–2187, 2022.

Authorized licensed use limited to: Hong Kong Polytechnic University. Downloaded on September 08,2026 at 01:38:46 UTC from IEEE Xplore. Restrictions apply.