Achieving Consistent Low Latency for Wireless Real-Time Communications with the Shortest Control Loop

Zili Meng[1,2], Yaning Guo[1], Chen Sun[2], Bo Wang[1],
Justine Sherry[3], Hongqiang Harry Liu[2], Mingwei Xu[1,4]
[1]Tsinghua University [2]Alibaba Group [3]Carnegie Mellon University [4]Zhongguancun Laboratory
zilim@ieee.org, gyn17@tsinghua.org.cn, qichen.sc@alibaba-inc.com, wangbo2019@tsinghua.edu.cn
sherry@cs.cmu.edu, hongqiang.liu@alibaba-inc.com, xumw@tsinghua.edu.cn

Abstract

Real-time communication (RTC) applications like video conferencing or cloud gaming require consistent low latency to provide a seamless interactive experience. However, wireless networks including WiFi and cellular, albeit providing a satisfactory median latency, drastically degrade at the tail due to frequent and substantial wireless bandwidth fluctuations. We observe that the *control loop* for the sending rate of RTC applications is inflated when congestion happens at the wireless access point (AP), resulting in untimely rate adaption to wireless dynamics. Existing solutions, however, suffer from the inflated control loop and fail to quickly adapt to bandwidth fluctuations. In this paper, we propose Zhuge, a pure wireless AP based solution that reduces the control loop of RTC applications by *separating congestion feedback from congested queues.* We design a Fortune Teller to precisely estimate per-packet wireless latency upon its arrival at the wireless AP. To make Zhuge deployable at scale, we also design a Feedback Updater that translates the estimated latency to comprehensible feedback messages for various protocols and immediately delivers them back to senders for rate adaption. Trace-driven and real-world evaluation shows that Zhuge reduces the ratio of large tail latency and RTC performance degradation by 17% to 95%.

CCS Concepts

• **Networks** → **Routers**; *Wireless access networks.*

Keywords

Real-time communications, congestion control, wireless network.

ACM Reference Format:
Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, Mingwei Xu. 2022. Achieving Consistent Low Latency for Wireless Real-Time Communications with the Shortest Control Loop. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3544216.3544225

1 Introduction

Real-time communication (RTC) applications like video conferencing [17, 47], cloud PC [42], and cloud gaming [8] are now prevalent in business and daily life. To provide a satisfactory user's experience, RTC applications require consistent low latency, even at the tail. For example, video conferencing demands consistent latency below 150ms [38], and cloud gaming demands under 96ms [39]. However, our analysis of a large-scale online live streaming platform with millions of daily active users (§2) shows that the median RTT of wireless users today is below 100ms (comparable to that of Ethernet users), but the 99th percentile *tail latency* is ~ 400ms. Intuitively, latency spikes at the 99th percentile indicate that RTC application users can experience one delayed video frame per every 100 frames (*i.e.* once every 5 seconds for a 20fps stream), which severely compromises user experience. Following this intuition, our measurements of the same system reveal that wireless users (including WiFi and cellular) encounter 2× more video rebuffering than Ethernet users.

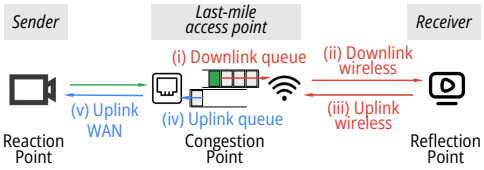


Figure 1: Control loop for rate adaption at the wireless last mile. Compared with existing solutions, Zhuge bypasses the segment (i) - (iii) to achieve the shortest control loop.

Transient congestion at wireless links is caused when available bandwidth for a user drops suddenly, *e.g.,* due to multi-user access and mutual interference. Available bandwidth of wireless networks can drop by 10× at the 99th percentile (§2.3). After such a sudden drop, packets quickly begin to queue at the AP, increasing end-to-end latency. Ideally, senders would react quickly when bandwidth reduction occurs, *e.g.,* by reducing their bitrate to prevent queue buildup, high latency, and loss. Unfortunately, we observe that senders are *fundamentally limited* in how quickly they can react, and it is *precisely when queues build up that senders react most slowly!*

The problem is that congestion signals are carried along the same congested path as data packets. Put simply, to observe that the bottleneck queue is filling, a sender must first receive an acknowledgement from a packet that has actually *waited in that queue.* Hence, congestion indicators like timestamps or losses take longer to reach the sender when the sender *most needs these indicators.* In Figure 1, we show the route taken both by data packets and the control signals they carry, in-band/explicitly (such as timestamps) or out-of-band/implicitly (such as their RTT).

Our key insight in this paper is that we can decouple the control loop from the full path that data packets traverse, hence protecting control signals from experiencing the full latency of filling, often buffer-bloated [60] queues. A carefully designed AP, on observing

Zili, Yaning, Bo, and Mingwei are with Institute of Network Sciences and Cyberspace, also with Beijing National Research Center for Information Science and Technology.

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544225

a filling downlink queue (i in Figure 1) can modify or delay packets in the uplink queue (iv in Figure 1), allowing congestion signals to reach the sender without the delay of the congested bottleneck.

Substantial research literature aims to improve network latency for wireless networks, but these approaches primarily succeed at improving *median* rather than *tail* latencies of RTC applications in the wireless network. We argue that the problem primarily stems from the fact that all of these approaches rely on a delayed control loop due to congestion signals needing to traverse the congested, high-latency path. For example, end-based solutions such as congestion control algorithms (CCAs) collect end-to-end signals (e.g., per-packet delays) at the sender to adjust the sending rate. However, one (inflated) control loop is still needed to collect the signals after sending a packet. Similarly, in-network solutions such as active queue management (AQM) create signals (e.g., packet drops) but these signals still have to be bounced by the receiver to the sender, which, again, suffers a long control loop.

While our key insight is straightforward, implementing it successfully in practice is challenging:

How can an AP predict packet latency for packets which have not yet been transmitted? Naïvely, an AP might simply measure the number of bytes queued in the downlink queue and divide by the available link capacity to measure a queuing delay. However, recall that link bandwidth is fluctuating (hence our problem) and so such an estimator is likely to be inaccurate.

How should the AP report the message back to the sender in a deployable way? A straightforward solution is enabling routers to directly transmit newly defined messages back to senders (e.g., XCP [41] or active network [25]). However, coordinating AP and senders that are usually maintained by different entities (§2.3) builds barriers for deployment at scale. Moreover, for existing deployed protocols at the sender, some use explicit signaling (e.g., timestamps) while others use implicit or out-of-band signaling (e.g., the RTT or RTT gradient). Some protocols react to a weighted moving average of the RTT [18]; some protocols are concerned with minimum RTT values over a particular window [12]; and some protocols react to inter-packet timings and are not concerned with RTT at all [19]. The AP must modify or delay upstream packets in a way that faithfully captures all of these factors, so that neither the sender nor the receiver requires modification.

Addressing these challenges, this paper presents Zhuge[1] that achieves consistent low latency[2] in wireless environments by minimizing the control loop. Zhuge includes a 'Fortune Teller' module that, on packet arrival at the downstream queue, makes a *prediction* as to that packet's delay to the receiver and back to the AP. The Fortune Teller separately estimates two factors influencing queuing delay (§4.1) and uses these to derive a combined prediction for every arriving packet. The second component of Zhuge is a 'Feedback Updater' which modifies upstream packets. Depending on the protocol, these modifications are based on either the raw packet delays recorded by the Fortune Teller, or *differences of packet delays* (details in §5.2) derived from the Fortune Teller.

We have implemented Zhuge in both simulation and with a WiFi-router based testbed (§7). Evaluation results with real-world wireless traces and configurations for both WiFi and cellular show that Zhuge improves key metrics on network conditions (*e.g.,* tail latency) and application performance (*e.g.,* video frame delay) by 17% to 95%. Further evaluation also shows that Zhuge is able to achieve satisfactory performance in the real world in different scenarios.

[1] Zhuge is a famous fortune-teller in ancient China.

[2] We mainly focus on recent CCAs that are designed to maintain a low latency, but fail to consistently achieve a low latency. Buffer-filling CCAs that suffer from a high RTT all the time (e.g., CUBIC [32]) are not our target.

Narayanan et al. (2020) [51]	Tail latency of the 5G hop does not improve against 4G, and could be around 200ms.
Daldoul et al. (2020) [22]	802.11ax (a.k.a. WiFi 6) has an average WiFi-hop latency of >30ms with 30 interferers.
Bhartia et al. (2017) [15]	More than one quarter of 802.11ac access points sufferfrom a latency of >100ms at the last hop.
Ghoshal et al. (2022) [30]	Maximum latency does not improve much for median users between 5G mmWave and 4G LTE.

Table 1: Recent measurement results of the wireless network latency.

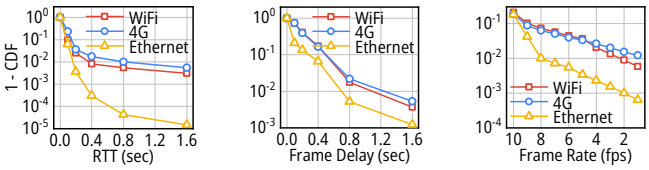


Figure 2: Comparison of RTT and video transmission quality of WiFi, 4G, and Ethernet according to data from **a large-scale online RTC application with O(1M) users every day**. Frame delay refers to the delay measured at the application layer.

2 Background and Motivation

In this section, we use real-world statistics to reveal the status of wireless tail latency (§2.1). Next, we analyze why existing solutions fail to achieve consistent low latency (§2.2). Finally, we present our motivation of reducing the control loop to ameliorate tail latency (§2.3).

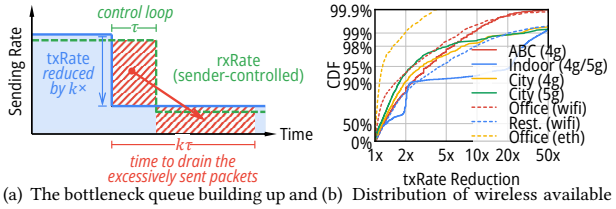
2.1 Understanding Wireless Tail Latency

We first answer the following two questions:

Why is the tail latency critical for RTC applications in wireless networks? Recent booming RTC applications not only require low latency in the median, but also demand consistent low latency at the tail. For example, suppose that most of the time, wireless users could experience a satisfactory RTT of <100ms. However, if the 99th percentile network RTT is >400ms, the network latency would far exceed the delay budget of applications [46, 50]. In this case, one frame out of 100 may suffer high latency, severely degrading user experience. Therefore, reducing tail latency is critical for RTC applications.

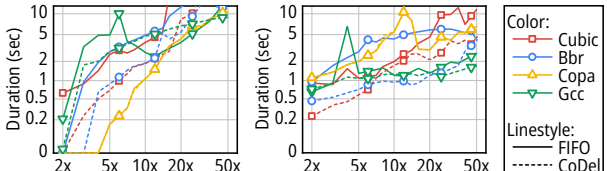
However, current wireless access network performance is not satisfactory at the tail. We back this argument with several observations. First, existing literature unveils the long tail latency even with advanced access technologies. We summarize measurement results in recent years in Table 1. Even with WiFi 6 (802.11ax) or 5G (mmWave), wireless networks still do not perform well. This is consistent with our investigation of content providers. "When experiencing network issues, plug your computer into a wired Ethernet connection if possible", stated in the guide of a cloud gaming provider [8]. Latency-sensitive applications turn out to prefer inconvenient but stable cable networks due to the high tail latency of wireless networks.

In addition, our own measurement results of an online real-time communication service, which serves millions of users every day (measurement methodology in Appendix A), also reveal degraded tail performance in wireless networks. We present the measurements



(a) The bottleneck queue building up and draining after ABW reduction. (b) Distribution of wireless available bandwidth reduction ratio.

Figure 3: The sudden drop of available bandwidth (ABW) could lead to transient increases of latency. Solid lines in Figure 3(b) are from open datasets [31, 49, 62] and dashed lines are measured on our testbed. ABW is the average value measured in a window of 200ms. Details in §7.2.



(a) RTT degradation. (b) CWND (sending rate) reduction.

Figure 4: The convergence duration after wireless bandwidth drop for different CCAs and AQMs. *RTT degradation duration is the time when RTT > 200ms. CWND rate reduction duration is the time for CCA re-convergence.*

of network conditions and application performance of Ethernet, WiFi, and 4G access networks. As shown in Figure 2, most of the time, wireless networks could provide a satisfactory RTT of <100ms. However, the 99th percentile RTT for wireless networks is as high as >400ms, which far exceeds the network latency budget for applications [46, 50]. The application-layer metrics also expose similar patterns: wireless users encounter 2× more video lags (long frame delay) than Ethernet users. Furthermore, the ratio of frame rate drops (video stalls) of wireless networks is 10× higher than that of wired networks.

Why does wireless latency fluctuate at the tail? The outstanding tail latency is caused by the transient mismatch of sending rate at the sender and available bandwidth (ABW) at the bottleneck queue. We illustrate the transient mismatch from the view of the bottleneck queue in Figure 3(a). When the ABW of one RTC flow suddenly drops by $k\times$ at the bottleneck router (the solid blue line), it takes one control loop τ for the CCA to reduce its sending rate (the green dashed line). During this period, the bottleneck queue still receives packets from the sender at its original sending rate. Thus, the queue builds up due to these excessive packets, as shown in the red shadow.

The duration of congestion is further amplified since it takes much longer to drain those excessively sent packets from the queue. Specifically, the packets that arrived at the bottleneck queue during the control loop τ would need $k\tau$ in total to be sent out. During this period, all packets sent out would experience an increased latency, degrading the user's experience.

Therefore, the transient increase of latency depends on (i) how violent the ABW fluctuates (k), and (ii) how soon the sender reacts (τ). As for the ABW fluctuation k, wireless channels are naturally more fluctuating than wired channels due to their variability. We calculate the available bandwidth every 200ms, during when the CCA should respond to such fluctuations, considering the RTT. from several open datasets and also our own measurements in the office and restaurant (details in §7.2). As shown in Figure 3(b), for all wireless datasets including 5G mmWave and 5GHz-band WiFi, 0.6-7.3% of ABW reduction rates are above 10×, which is much higher than the <0.1% of wired networks. As for the control loop τ, in most cases, the congestion controller needs one RTT to adjust the sending rate upon receiving the congestion signals (e.g., increased delay, packet losses). When the bottleneck queue starts to build up, the end-to-end RTT also inflates, further preventing the congestion signals from reaching the sender. Consequently, the end-to-end latency will fluctuate at the tail.

2.2 Existing Solutions

The reduction of ABW (k) is due to contention in the link layer and below [40] and is unavoidable in most time (e.g., due to wireless interference). Many transport layer innovations have been proposed to improve the steady state median latency of a connection. For example, BBR [18] moves the working point of congestion control from a full queue in CUBIC [32] to an empty queue. CoDel [52] queue management also tries to shorten the queue in the steady state in a variety of network conditions compared with FIFO. Subsequent research efforts (including congestion control [12, 19, 24] and active queue management [34]) further provide insightful thoughts of maintaining the optimal working point with different feedback signals. Standing on the shoulders of giants, the median latency for applications can be nicely controlled. However, they are insufficient to reduce the tail latency, which we will analyze below.

End host-based solutions. For network layer and above, existing end host-based solutions fail to quickly adapt to the ABW reduction due to their long and inflated control loops. Recalling Figure 1, when the green shaded packet arrives at the congestion point and observes a long queue, it first needs to go through the queue (i), transmitted to the receiver (ii), the corresponding feedback delivered from the receiver to the access point (iii), and finally sent to the sender (iv and v). Since the shortest time for the sender to be notified is one full control loop including segments (i)-(v), a pure end host-based CCA cannot timely adapt to transient bandwidth fluctuation. We further simulate the performance of recent latency-sensitive CCAs (BBR [18], Copa [12], and GCC [19]) together with AQMs in Figure 4. When the ABW is reduced by 10× or more, all these algorithms, working with or without latency-aware AQMs, suffer from seconds of RTT degradation. The inflated control loop for end host-based solutions results in severe wireless queuing.

In-network solutions. Solutions modifying in-network devices also fail to timely feed back these signals. For example, AQM such as CoDel [52] drops the packets in the front of the queue to reduce the downlink queuing latency (i) in Figure 1, yet still suffers long wireless latency (ii) and (iii), which could be more than 100ms [15]. Moreover, AQMs are mostly designed to *drop* some packets, while many modern CCAs are designed to be responsive to the increase of packet *delay* and insensitive to packet drops [12, 18, 19]. This can also be validated in Figure 4(a): CoDel can hardly improve the performance of delay-based CCAs such as Copa. There are also a line of solutions to co-design the hosts and in-network routers for decades to achieve better feedback from the network, including XCP [41], RCP [58], Kickass [26], and ABC [31]. However, their design goals are getting a precise estimation of network conditions from routers, while the gathered information still needs to go through the full control loop. We also compare the performance of Zhuge against ABC to demonstrate the potential room for improvements with host-router

co-design and our further improvements in §7.

2.3 Our Proposal: Reducing the Control Loop

Our key insight to reduce wireless tail latency is to separate the congestion feedback from the congestion by sensing the network conditions as early as possible, timely carrying the conditions back to the sender to *minimize the control loop*, and performing the above operations in a deployable way.

The earliest signal – one packet knows its fortune upon arrival. In most cases, when one packet arrives at the bottleneck queue, it can predict its delay with visibility of the entire queue. For example, the queuing delay for the packet could be roughly estimated by dividing the queue length with the dequeuing rate. Therefore, when the dequeuing rate decreases, we can observe increasing queuing delay upon the arrival of subsequent packets. Compared with other consequent signals such as the packet loss or the measured queuing delay, the estimated queuing delay is the earliest signal for the reduction of ABW. Therefore, we are motivated to utilize this earliest signal to timely control the sending rate and adapt to ABW reduction.

Quickly delivering the earliest signal back to the sender. Merely finding the ABW reduction signal is not enough. We need to quickly carry this signal back to the sender. An ideal solution is directly *telling* the sender from the bottleneck queue about its current status. In this way, such a signal could bypass the inflated part of the control loop (downlink queuing (i), downlink wireless transmission (ii), and uplink wireless transmission (iii) in Figure 1). Meanwhile, the latency of the uplink queue at the AP (iv) and the latency of WAN (v) is usually stable. The uplink of the AP is often the Ethernet connection to the Internet, usually with hundreds of Mbps capacity. The WAN latency (v) is the latency between the last-mile AP and the sender. The Ethernet users will also suffer these two parts of control loop, which are relatively stable according to our Ethernet measurements in Figure 2.

Patching the last-mile router only might be deployable. Reviewing the history of transport layer designs, there are a series of excellent efforts that unfortunately are not widely deployed due to practical issues. For example, XCP [41], RCP [58], Kickass [26], ABC [31], and active network [25] in recent two decades all require modifications on both the server and some or all routers. However, servers are usually controlled by content providers (e.g., Google, Facebook), while routers by vendors (e.g., Netgear for APs). Coordinating all these parties to push a new transport innovation forward is extremely challenging, if not impossible. Different from above work, Zhuge patches the last-mile AP only, which could reduce the barrier to deploy at scale. AP vendors could individually implement and observe the performance benefits without co-operation with content providers. Moreover, from the view of home users, the last-mile AP is the only place they can control if they seek a better performance. We are thus motivated to limit the modifications to the last-mile to make Zhuge deployable at scale.

3 Zhuge Design

This section presents the design challenges and framework overview of Zhuge to control the wireless tail latency.

3.1 Design Challenges

Zhuge handles wireless tail latency by reducing the control loop. However, Zhuge design confronts two major challenges.

Timely and precise estimation of packet latency for RTC traffic. Zhuge estimates the future latency of a packet upon its arrival at the wireless last mile to obtain network conditions as early as possible. A per-packet precise estimation is necessary to properly guide CCAs in the sender for rate adaption. However, precise latency estimation is challenging for RTC traffic in wireless environments, as the bottleneck queue is in a transient fluctuation at a *sub-RTT granularity*, due to two reasons.

- *Bursty packet arrivals of RTC traffic.* RTC applications generate contents in the unit of a video frame. To reduce the end-to-end latency, senders tend to burstily send packets of the same frame out [21]. This indicates that the queue might build up very quickly even in the steady state.
- *Bursty packet departures of wireless channel.* The sharing nature of wireless networks results in the contention of wireless channel resources and frequent bandwidth fluctuation. Wireless protocols tend to aggregate several packets into one MAC frame (e.g., aggregated MAC protocol data unit, or AMPDU, in WiFi) to compromise wireless contention. In this case, tens of packets might be aggregated into one AMPDU and dequeued simultaneously.

A naive estimation approach is simply dividing the queue length by the dequeuing rate. However, this approach is faced with a *transience-equilibrium nexus* [45]: The dequeuing rate is usually measured over a sliding window (e.g., 40ms for WiFi in [31]). A short window would lead to the variability of measurement during the steady state, while a long window misses transient latency fluctuation at sub-RTT granularity. Thus, it is challenging to timely and precisely estimate the per-packet latency for RTC traffic at the wireless last mile.

Effective message feedback for various protocols and CCAs. Zhuge notifies the sender with the estimated wireless network conditions as quickly as possible. A straightforward solution is constructing a new type of feedback packets to the sender. However, for most CCAs deployed in the wild, network conditions such as the current available bandwidth are not explicitly delivered on the Internet. Directly telling the network conditions to the sender would need modifications at the sender simultaneously to make the message understandable to the CCA. As mentioned above, we prefer an AP-based solution without modifying the sender for deployability at scale.

Making this challenging, transport protocols and CCAs adopted by real world applications are highly diversified. The headers of transport protocols could be unencrypted (e.g., TCP) or encrypted (QUIC). To achieve lower latency, RTC applications prefer to customize CCAs, which rely on different signals to adjust the sending rate. For example, some of them modify the TCP CCA in the kernel [5]. For WebRTC-based applications, network conditions are periodically summarized into a special feedback packet [55]. Various CCAs make it challenging to effectively deliver the network conditions to the sender.

3.2 Framework Overview

In response to the above challenges, we design two building blocks in Zhuge: a *Fortune Teller* and a *Feedback Updater*.

To achieve timely and precise prediction of packet latency, we introduce the Zhuge Fortune Teller in §4 to tell the fortune (future latency) of each packet upon its arrival. To overcome the transience-equilibrium nexus and faithfully obtain precise per-packet latency, we break the latency into different parts and introduce long-term and short-term estimators. We measure the average dequeuing rate

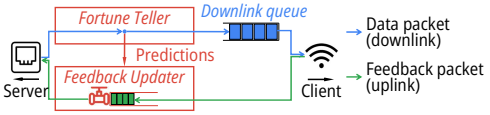


Figure 5: The overall workflow of Zhuge at the last-mile AP. Zhuge contributes the Fortune Teller and Feedback Updater.

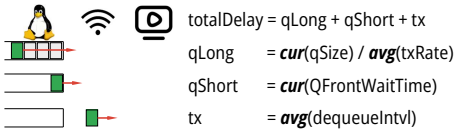


Figure 6: Different delay components that the Fortune Teller will estimate. `qLong` and `qShort` together form the queuing delay at the network layer. `tx` is the transmission delay at the link layer.

to calculate the long-term queuing delay, and the packet sojourn time at the front of the queue to respond to short-term fluctuations.

To effectively notify the sender with the latest conditions, we present the Zhuge Feedback Updater in §5 to convert predicted network conditions to signals that senders can understand. We categorize existing protocols in RTC applications into out-of-band feedback and in-band feedback. For out-of-band feedback protocols, the arrival of feedback packets are signals to the sender (e.g., ACK packets in TCP). In-band feedback protocols carry network conditions in the payload of feedback packets, such as the transport-wide congestion control feedback (TWCC-FB) packets in WebRTC [35]. Accordingly, Zhuge designs different feedback mechanisms to carry the latency back to the sender for a variety of protocols.

The overall workflow of Zhuge is presented in Figure 5. When a packet arrives at the wireless access point via the Ethernet port, Fortune Teller would predict its fortune and also forward the packet as usual to the downlink queue. Feedback Updater will then update the estimation into the feedback packets in the reverse direction. If a newly arrived packet observes a degraded network condition (e.g., increasing queue length), estimated wireless latency could be immediately applied to feedback packets in the reverse direction of the same flow. In this way, the earliest signals could be carried back to the sender, bypassing the queuing delay and wireless transmission delay of the control loop (part (i)-(iii) in Figure 1).

4 Fortune Teller

Telling the fortune of a packet is to predict when it will arrive at the client, *i.e.,* the subsequent delay it will experience. In a wireless network, such delay can be decoupled into two segments [33], including (i) Queuing delay: the delay between the packet arriving at the access point, and the packet leaving the queue disciplines to the underlying driver (i.e., the delay in the network layer). (ii) Transmission delay: the delay between the packet being passed to the wireless driver, to the time it arriving at the receiver (i.e., the delay in the link layer). Next we introduce how to timely predict these two delays respectively.

4.1 Queuing Delay Prediction

As discussed in §3.1, the strawman solution of dividing the queue size by the dequeuing rate confronts the transience-equilibrium nexus. A short sliding window will lead to drastic fluctuations of the predicted delays due to the bursts of arrivals and departures, and a long window will fail to quickly detect the change of network conditions. In

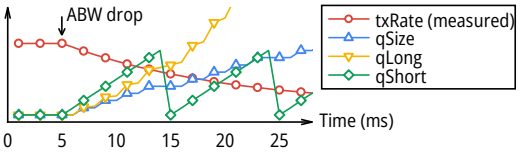


Figure 7: How `qLong` and `qShort` react to the ABW drop at 5ms.

response, we analyze how to capture the latency fluctuation incurred by the two reasons respectively.

- *Bursty packet arrival of RTC traffic.* The bursty RTC traffic quickly builds up the wireless queue. Our design choice is to predict the packet fortune for each packet instead of on a periodic basis. In this way, the delay differences within a burst of RTC traffic can be captured by taking the queue size observed by each packet as input.
- *Bursty packet departure of wireless channel.* Bursty packet departure introduces transient glitches to the dequeuing rate at the millisecond timescale, which is easily averaged and therefore missed with existing sliding window-based measurements. Our main observation is that when the dequeuing rate is suddenly reduced, an instantly measurable signal is the *waiting time of the packet at the front of a queue* (denoted as the front packet). For example, when the channel starts to become busy, the packet at the front of the queue has to wait for more time to get a chance to be transmitted.

Since the causes of delay are different when the packet is at the front of the queue and is not, we decouple the queuing delay into two parts: long-term queuing delay (`qLong`) and short-term queuing delay (`qShort`), as shown in Figure 6. Specifically, `qLong` is defined as the delay from the time when one packet arrives, to the time when that packet is at the front of the queue, which is used to cover the latency fluctuation induced by wireless contention and bursty RTC traffic. We could estimate `qLong` as the ratio of current queue size over average dequeuing rate since it's more affected by the queue dynamics. Short-term queuing delay is the time from the time one packet is at the front of the queue, to the time when that packet is finally dequeued. `qShort` is more related to the sending pattern at the link layer (e.g., the aggregation of MAC data units will lead to fluctuations in qShort). We therefore individually predict `qLong` and `qShort`, and take their sum as the estimation of queuing delay. In Figure 6, $avg(\cdot)$ denotes the *average* value over a sliding window, while $cur(\cdot)$ denotes the *current* value measured at the time of calculation. $qSize$ is the size of the queue, $qFrontWaitTime$ is the time that the current front packet of the queue has waited so far, and $txRate$ is the dequeuing rate of the queue.

Using the combination of long-term and short-term queuing delay prediction has two advantages. We illustrate the advantages with an example in Figure 7. First, using `qShort` can quickly detect the ABW drop. When the ABW starts to decrease, since the queue needs some time to build up, and the measured txRate also needs some time to decrease due to the sliding window, `qLong` increase slowly. Instead, packets have to wait for longer time to send, which could be immediately observed. As illustrated in 5-15ms in Figure 7, `qShort` would dominate the increase in total queuing delay, quickly reflecting the ABW drop. Second, using `qLong` could provide a stable and accurate estimate of the queuing delay when the queue has already been built up. For example, when the ABW while the bottleneck queue is still overloaded (e.g., after 15ms in Figure 7), `qLong` would dominate the queuing delay, providing a stable and accurate estimation.

Next, we further introduce how we handle two practical issues in realizing the estimation of queuing delay.

Adjustments against bursty departure. The bursty departure of the queue due to the aggregation of packets at the link layer could affect the accuracy of the estimation of qLong: when there are several packets in the queue, they may be sent out together at once. In fact, according to our design, fluctuations within a burst should be reflected on qShort. Thus, when calculating qLong, we estimate *qSize* as

$$\text{qSize} = max(\text{sizeOfPacketsInQueue} - \text{maxBurstSize}, 0) \quad (1)$$

where *maxBurstSize* is the maximum size of simultaneous packet departures at the resolution of 1ms.

Calculation with queue disciplines. Another issue in practice is that queues in reality might not be FIFO as assumed in research papers [31]. For example, the default queue discipline in systemd has been changed to fq_codel among different flows differentiated by their 5-tuples [3]. For cellular networks, each flow also has its own queue isolated from competing flows [31]. In these cases, we need to calculate the statistics of the RTC flow's corresponding queue.

4.2 Transmission Delay Prediction

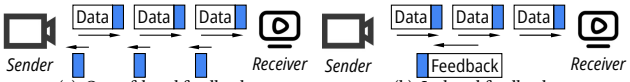
In this paper, we mainly target at the estimation of delays in the WiFi network. We refer the readers to [31] for the estimation on cellular networks. Predicting the transmission delay for each packet is challenging since it is correlated to the underlying wireless drivers and physical channels. Especially for high-performance wireless devices (e.g., 802.11ax), critical features (e.g., bit-rate selection and frame aggregation) are coded in the hardware device and inaccessible from the access point CPU without significant vendor interaction [15]. For example, many Netgear routers adopt the Qualcomm Atheros hardware [1], where performance-critical features (frame aggregation, etc.) are hard-coded and inaccessible. Therefore, it is challenging to predict the transmission delay of the wireless channel.

According to [31], we summarize the following observations of the transmission delay. First, similar to all link layer protocols, there should be only *one data unit* in transmission in the wireless channel. For example, an 802.11ac sender might aggregate several packets into one data unit (aggregated MPDU, or AMPDU). However, multiple AMPDUs cannot be transmitted simultaneously since their signals will interfere with each other. Therefore, the wireless driver will aggregate several packets into one AMPDU, send it out, and wait for acknowledgment or timeout of that AMPDU. Second, with recent efforts in the Linux mainline, the queue in the lower layers of the wireless network stack has been exposed to the queue discipline [33]. In this case, the lower layer queue in the wireless network stack is only used to aggregate multiple packets into a link layer frame.

Consequently, as shown in Figure 6, the transmission delay tx is calculated as the average interval between packet departures from the network layer queue, with a window similar to txRate. The sliding window should be long enough to cover at least two bursts from the sender so that packets are continuously measured. Note that since multiple packets might be aggregated and dequeued simultaneously, we do not calculate the intervals that are less than one millisecond.

5 Feedback Updater

Zhuge delivers the estimated latency back to the sender in a message that is comprehensible to the sender. To avoid modifications at end hosts, Zhuge abide by the original feedback message format of application protocol and CCAs. This section starts by categorizing feedback mechanisms of popular CCAs for RTC applications (§5.1), and then introduce our corresponding solutions (§5.2 and §5.3).



(a) Out-of-band feedback. (b) In-band feedback.

Figure 8: Out-of-band feedback protocols do not explicitly carry the feedback information in the payload while in-band ones do. Blue and white blocks denote packet headers and payloads.

	Protocol	CCA	Application
Out-of-band (§5.2)	TCP QUIC [36]	PCC [24] BBR [18] Copa [12]	Meta Live [29] Windows 365 [42] Twitch [56] Tencent Start [5]
In-band (§5.3)	RTP+RTCP [55]	GCC [19] NADA [65] Scream [37]	Google Stadia [23] Zoom [47] Microsoft Teams [54]

Table 2: We categorize the feedback mechanisms of existing RTC applications into out-of-band feedback and in-band feedback. Protocols of some applications are identified by ourselves.

5.1 Feedback Mechanism Classification

We investigate popular RTC applications and summarize their feedback mechanisms in Table 2. They can be categorized into two types, in-band and out-of-band. We present their behaviors in Figure 8.[3]

- *In-band feedback.* As shown in Figure 8(b), in-band feedback means that the feedback information is explicitly written in the payload of a specific type of feedback packets. For example, the Real-Time Protocol (RTP), together with the Real-Time Control Protocol (RTCP), follows the in-band feedback. The receiver records the time of arrival of each data packet and periodically constructs a feedback packet to carry time intervals back to the sender [35].
- *Out-of-band feedback.* Out-of-band feedback mechanisms do not explicitly write the information related to rate control in the payload of feedback packets. In contrast, the sender calculates all network conditions itself upon receiving the feedback packets. For example, a TCP client will acknowledge each packet it receives. When the sender receives the ACK packet, it will then calculate the RTT, receiving rate, and other network conditions.

We separately design solutions for the above two different feedback mechanisms. For out-of-band feedback mechanisms, network conditions are measured at the sender only. Our observation is that we can *deliberately delay* the feedback ACK packets to carry the network conditions back. For in-band feedback mechanisms, as feedback information is written in the payload of feedback packets, we need to update the payload of feedback packets. Next we introduce two solutions in detail.

5.2 Out-of-band Feedback: Delaying ACKs

ACK packets are used as messages for applications relying on out-of-band feedback, but are consumed in different ways by various CCAs. For example, BBR counts the receiving rate and queries the minimal RTT of ACK packets for rate adaption, while Copa [12] is sensitive to

[3] Some protocols may utilize both feedback mechanisms. For example, the RTP sender also measures the RTT itself, similar to TCP [55]. This RTT information is not used for rate control, but is only used to stabilize the control loop in RTP.

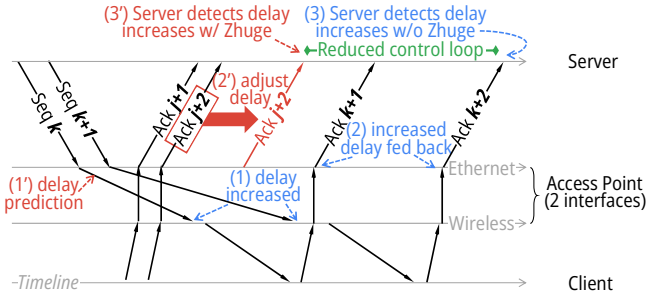


Figure 9: Zhuge immediately delays the feedback packets in the reverse direction to carry the predicted fortunes back.

per-packet delay. To satisfy the requirements of different CCAs, our design goal is to faithfully deliver the estimated latency in the finest per-packet granularity by *delaying ACK packets*. CCAs can then aggregate fine-grained information and react in their own ways.

We present an illustration of how Zhuge carries the predicted packet fortunes back from the view of AP in Figure 9. Blue arrows indicate how network conditions can be sensed by the sender without Zhuge. Assume packets with sequence numbers k and $k+1$ arrive at the AP from the server, and now the available bandwidth drops. Without Zhuge, the packet behind (seq $k+1$) will be dequeued later than expected, and the queuing delay will gradually increase ((1) in blue). The client will then receive these two packets with an enlarged interval, and consequently acknowledge them with that interval. The ACK packets will then arrive at and depart from the AP with an enlarged interval ((2) in blue). As shown in Figure 10, without Zhuge, the sender can only acknowledge increased RTT when the ACK of delayed packets arrives at time *deltaDelay*.

With Zhuge, the latency of packets seq k and $k+1$ could be predicted upon their arrival ((1') in red). If the Fortune Teller predicts that the delay is increasing, we can immediately delay earlier ACKs of previous packets that have arrived or will arrive at the access point. As illustrated by red arrows in Figure 9, we can deliberately enlarge the interval between other ACK packets (ACK $j+1$ and $j+2$) to timely notify the sender ((2') in red). In this case, the server can detect the available bandwidth drops when packets with the adjusted delay arrive at the server ((3') in red). The RTTs of different packets measured by the server with Zhuge would then be shifted forward as shown in Figure 10. Consequently, the control loop of CCAs is reduced by $(k+1)-(j+1)$ (counted in ACK number, the green arrow in Figure 9). Also note that, Zhuge does not need to look at and match the sequence and ACK number – the numbers presented here are for illustrative purpose. Instead, Zhuge only looks at the 5-tuple to identify flows, and views the sequence and ACK streams as blackboxes. In this way, Zhuge could still work even the transport protocol is encrypted (e.g., QUIC).

However, downlink data packets and uplink feedback packets arrive at the AP asynchronously. Thus, it is often impossible to one-on-one map the delay predicted by the downlink data packets to the uplink feedback packets. When packets arrive, the Fortune Teller will be updated according to current network conditions. The updated queue conditions include the qLong, qShort, and tx, as introduced in §4. The final predicted total delay is calculated as:

$$\text{totalDelay} = \text{qLong} + \text{qShort} + \text{tx} \tag{2}$$

Below we introduce design principles of Zhuge to ensure the precision of latency of packets.

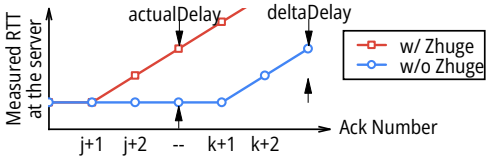


Figure 10: Zhuge shifts the curve of RTT forward by delaying earlier returning ACK packet to quickly feedback network conditions. The *actualDelay* is the control loop of Zhuge.

Delivering precise long-term latency in the steady state. Since Zhuge deliberately delays the feedback packets in the uplink, a natural concern is whether such a delay will affect the estimation of network RTT in the steady state. For example, for the packet seq $k+1$ in Figure 9, it has already suffered a long queuing delay in the downlink direction. If Zhuge also introduces a non-trivial delay for its feedback ACK packet ACK $k+2$ in the uplink direction, it will exaggerate the real RTT and might interfere with the estimation of CCAs.

To handle this problem, we do not directly add the *absolute* estimated delays from the downlink direction into the additional ACK delay in the uplink direction. Instead, we record the *relative delay deltas*, *i.e.* the delay difference between consecutive downlink packets. When the estimated delay is increasing, we could record a series of positive delay deltas from the downlink direction and gradually increase the delay in the uplink direction. When the queue has already been steadily built up (e.g., for packets after seq $k+1$), the delay delta would be around zero, and the feedback packet in the uplink direction would not suffer from additional delays.

Delivering precise short-term latency fluctuation. Short-term per-packet latency dynamics are vital for latency-sensitive CCAs like Copa. These CCAs will utilize the patterns of packet delays at the sub-RTT level to control the sending rate. However, naively leveraging the delay delta mechanism may not faithfully deliver short-term latency fluctuations. The reason is that short-term latency varies packet-by-packet. Not every delay delta can be carried in one separate ACK. This might result in the accumulation of multiple delay deltas into one ACK, which is unfaithful. For example, when three data packets arrive at the AP with delay deltas of +1ms between each packet, directly delaying the next ACK for +3ms would introduce a sharper delay increase than the actual value.

To address this problem, instead of delivering per-packet delay delta, our key idea is pursuing the *distributional equivalence* between downlink delay delta and uplink ACK delays. We maintain a distribution of recent delay deltas of the downlink data packets. Upon the arrival of a downlink packet, we calculate the delay delta according to the predicted delay by the Fortune Teller. When an uplink feedback packet arrives at the access point, we *sample* the distribution of recent deltas, and use the obtained value to delay the feedback packet. In this case, even under bursty packet arrival and departure, Zhuge is able to mimic the delay distributions to the feedback packets.

Preserving the order of feedback packets. Our approach of applying delay deltas to uplink feedback packets introduces an additional challenge of order preserving of feedback packets. For example, if packet ACK $j+1$ and $j+2$ arrive simultaneously, and ACK $j+2$ samples a lower delay than ACK $j+1$, the AP may send ACK $j+2$ in front of ACK $j+1$, which leads to out-of-order of feedback packets and confusion at the sender. Clamping the sending time of the

Algorithm 1: On data packets: Out-of-band feedback

```
1  deltaDelay = curTotalDelay - lastTotalDelay
2  if deltaDelay ⩾ 0 then
3      deltaHistory.push_back(deltaDelay)
4  else
5      tokenHistory.push_back(-deltaDelay)
6  lastTotalDelay = curtotalDelay
```

Algorithm 2: On ACK packets: Out-of-band feedback

```
1  actualDelay = min (0, lastSentTime - curArrvTime)
2  actualDelay += random(deltaHistory)
3  while tokenHistory is not empty do
4      if tokenHistory.front > actualDelay then
5          tokenHistory.front -= actualDelay
6          actualDelay = 0
7          break
8      else
9          actualDelay -= tokenHistory.front
10         tokenHistory.pop_front
11 Schedule to send the current ACK packet after actualDelay
12 lastSendTime = curArrvTime + actualDelay
```

subsequent packets to the precedent ones, such as holding ACK j+2 until ACK j+1 has been sent, will lead to the overestimation of RTT.

In response, we introduce a delay token to preserve the order of feedback packets and also avoid the overestimation of RTT. When we need to let the subsequent feedback packets wait for the sending of precedent packets, we store the waiting time as a delay token. Next time when a positive delay delta is sampled, we will first try to consume the token. In this case, the average values of actual delays will be maintained the same as the predicted delays.

We finally present the workflow of how Zhuge Feedback Updater uses the predicted fortune to update the feedback packets. As shown in Algorithm 1, upon arrival of each data packet, given the predicated delay of that packet, Zhuge first calculates the delay delta (line 1). If the delta is nonnegative, we store it into a sliding window. Since Zhuge can only delay the ACK packets with a positive time, if the delta is negative, we need to store it as tokens (line 4-5). Asynchronously, upon arrival of each ACK packet, Algorithm 2 will be executed to properly delay ACKs. curArrvTime is the arrival timestamp of the current ACK, and lastSentTime is the calculated timestamp to send the last ACK packet from the AP to the server. For order preservation, Zhuge first calculates the minimum delay for the current ACK packet to make sure that the current ACK packet would be sent after previous ACK packets (line 1). Zhuge then randomly samples a delay delta from the recent deltas in a sliding window (line 2). Zhuge further checks if there are outstanding tokens and consumes the tokens if available (line 3-10). Finally, the current ACK packet will be delayed and sent after actualDelay (line 11).

5.3 In-band Feedback: Updating Payloads

For in-band feedback mechanisms such as RTCP [55], the feedback information (e.g. per-packet receiving time) is written in the payload of feedback packets. We need to update their payloads to carry the freshly estimated latency back to the sender. We use the RTP (data)/RTCP (feedback) protocol pair to introduce how we update the feedback packets with two steps.

- *Step 1: Packet fortune recording.* Upon the arrival of each RTP packet, Zhuge will predict its fortune and then store the predicted delay together with its RTP transport-wide congestion control (TWCC) sequence number in the RTP header.
- *Step 2: Feedback construction.* When it's the time to feedback the current network conditions back to the sender (e.g., once per RTT or per frame [35]), Zhuge will behave like the RTP receiver and construct a TWCC feedback packet based on stored delays and sequence numbers. To ensure timestamp consistency, Zhuge will only send the TWCC packets constructed by itself and drop all TWCC from the client. For other types of feedback packets (e.g., negative acknowledgement for loss recovery, receiver reports, etc.), Zhuge will forward it from the client to server as normal.

Detailed RTP/RTCP packet formats are presented in RFCs [35, 55]. Meanwhile, there are two practical concerns regarding the implementation of Zhuge in-band feedback mechanism.

Time synchronization. Since the timestamps on the AP may not be synchronized with the receiver, a straightforward concern is whether the time differences between the AP and the receiver would affect the estimation of CCAs. In fact, the server is designed to tolerate the time differences between the server and the constructor of feedback packets (no matter clients or APs) since the server is not synchronized with the client either. Therefore, the timestamps of produced TWCC packets are from the same AP clock and consistent with the server.

End-to-end encryption. In some cases, RTP data packets and RTCP feedback packets might be end-to-end encrypted [14]. Zhuge could work in such cases due to the following reasons. First, Zhuge does not need to decrypt the RTP data packet payload. Instead, Zhuge only needs to record sequence numbers, which are explicitly readable in the header. Second, Zhuge does not need to decrypt the RTCP feedback packet payload either. Zhuge only needs to *encrypt* the constructed feedback packet so that the server can correctly decode the packet. Fortunately, in some cases in practice, server and client share the public key in plaintext with each other at the beginning of the connection [14]. Zhuge might intercept and save the public key of the server, and use it to encrypt the constructed feedback.

6 Discussion

Here we discuss some practical considerations in the deployment of Zhuge, as well as the limitations.

Last-mile *v.s.* first-mile. We mainly introduce and evaluate the performance of Zhuge in the direction of downlink, where the wireless network serves as the *last-mile*. This is because for many RTC applications such as remote desktop, cloud gaming, and video-on-demand, videos are disseminated from servers to clients. Remote servers as senders adjust the sending rate and suffer from a long control loop. For other peer-to-peer RTC applications, such as video conferencing, the wireless network as the *first-mile* might also introduce tail latency. In this case, queues are built up in the clients. Mechanisms in Zhuge can also be used to handle first-mile tail latency by manipulating the client-side network stack, which needs integration with the application and is beyond our scope.

Fairness. Reducing the control loop for a CCA indicates a faster reaction to network conditions, which might imply a greater aggressiveness in both sending rate increase and decrease. A natural

concern is whether Zhuge impairs the fairness between optimized flows and other ones. Our answer is *no* because Zhuge does not prioritize target flows by sacrificing others. (1) When *sending rate increases*, wireless queue should be near empty. In this case, flows optimized by Zhuge have a similar control loop to those without Zhuge and will not become more aggressive. (2) *Sending rate decrease* may be caused by wireless queues building up. Zhuge merely reduces the control loop and accelerates convergence, while the converged fairness between different CCAs should be handled during the design of CCAs [48]. We further evaluate the fairness of Zhuge in §7.6.

Scalability to new protocols. In this paper, we propose solutions for a wide range of applications as long as they use the TCP, QUIC or RTP/RTCP protocols. However, new protocols may evolve in the future. For new out-of-band protocols, as long as we could identify the flow information from packets, Zhuge could still work from the network layer. For example, since we do not need to know the specific sequence numbers of the packets, even QUIC encrypts all packets end to end, Zhuge is still able to work with QUIC. For in-band protocols, we need operators to release the format of the protocols to accordingly modify the Feedback Updater in Zhuge.

7 Evaluation

We first introduce our implementation of Zhuge in §7.1 and the experimental setup in §7.2. Then, we evaluate the performance of Zhuge to answer the following questions:

- *Can Zhuge improve the tail performance under real-world wireless traces?* We evaluate Zhuge over RTCP/RTP and TCP with five real traces. Evaluation shows that Zhuge can reduce the ratio of long tail latency by up to 75%, and improve the application performance by up to 91%. (§7.3)
- *How does the performance of Zhuge vary under different types of wireless competition?* We craft wireless scenarios of bandwidth reduction, flow competition, and wireless interference. We observe performance improvement of Zhuge under all scenarios. (§7.4)
- *How much performance improvements Zhuge can bring in the real world?* Our prototype deployment of Zhuge in our office environments shows that Zhuge could improve both the network and the application metrics from 17% to 94.7%. (§7.5)
- *What is the overhead of Zhuge in terms of steady state performance, fairness, and CPU resources?* We find that Zhuge does not compromise the steady-state bitrate of RTC flows, fairness with other flows, and has acceptable overhead. (§7.6)

7.1 Implementation

We implement Zhuge with both NS-3 *simulator* and a *testbed* based on production wireless APs. In our simulation, we implement a simplified video encoder and decoder according to reference implementations in WebRTC. We implement both the RTP/RTCP and TCP protocol stacks, as well as advanced CCAs and AQMs listed in §7.2. We construct network layer and link layer wireless queues, and implement Zhuge for simulation. We set the sliding window to 40ms in the Fortune Teller and Feedback Updater since our video stream is at 25fps. For testbed experiments, we implement Zhuge in OpenWrt, an open-source operating system for embedded network devices. The Fortune Teller and Feedback Updater are implemented as user-space features in OpenWrt that use packet sockets to observe and modify packets. We identify target RTC flows by matching its IP with a configurable IP list maintained in Zhuge [7, 10]. We use a Netgear WNDR 3800 router [1] that runs OpenWrt and supports WiFi 802.11n for performance evaluation. We also deploy Zhuge on a TP-Link router to measure CPU resource overhead.

7.2 Experimental Setup

We produce videos at 1080p 24fps with an average bitrate of 2Mbps. Below we present baselines, traces, and metrics we use.

Baselines. Zhuge can work with advanced CCAs and active queue management (AQM) mechanisms. In our evaluation over RTP/RTCP, we implement the following solutions:

- *Gcc+FIFO.* Google Congestion Control (Gcc) [19] is the default CCA of WebRTC and is adopted by many applications such as Google Stadia and Google Meet. GCC is sensitive to both packet loss and increased network latency. Thus, we choose Gcc as the CCA for the RTP/RTCP protocol, and use the FIFO scheduler in wireless queues as a baseline.
- *Gcc+CoDel.* CoDel [52] is an AQM mechanism designed to handle bufferbloat. It would drop packets in the front, instead of tail, of queue when the queuing delay increases to timely deliver the congestion signal to senders.
- *Gcc+Zhuge (+CoDel).* We implement Zhuge over RTP/ RTCP and evaluate the performance when working with Gcc.

For TCP evaluation, we implement the following solutions. Note that the CCAs we choose are loss-insensitive. Thus, to be concise, we evaluate each solution with FIFO and CoDel respectively, and select the better performer as the baseline.

- *Copa.* Copa [12] is a latency-sensitive CCA for TCP. It can achieve low latency according to many experiments [11, 31] and is already deployed in real streaming services [29].
- *Copa+FastAck.* FastAck [15] is a WiFi AP-based optimization that reduces latency by counterfeiting a TCP ACK packet on receiving the 802.11 ACK from the client device.
- *ABC.* ABC [31] optimizes wireless network performance through network-host coordination. It detects the network conditions directly from the access point, and reports them to the sender. However, ABC needs to modify the wireless access point, the client, and the server simultaneously.
- *Copa+Zhuge.* We implement Zhuge over TCP and evaluate the performance of Zhuge when working with Copa.

Traces. We use five real-world traces with sub-second resolution. Two are from WiFi networks and three from cellular. The traces record the bandwidth and delay at each timestamp.

- *W1 - Restaurant WiFi.* We measure the goodput of a public WiFi AP provided by a crowded restaurant [6] for 3 hours during dinner, and calculate the goodput at the resolution of 200ms. The WiFi AP operates in 2.4GHz with 802.11ac. We leave the measurement details to Appendix A.
- *W2 - Office WiFi.* We also measure the goodput of the WiFi AP in our office for 10 hours in the office hour. Our office APs operate in the 5GHz band with 802.11ac.
- *C1 - Indoor Mixed 4G/5G.* Goodput is measured over both 4G and 5G cellular networks in an indoor scenario [49].
- *C2 - City 4G* and *C3 - City 5G.* Literature [62] collects packets over both 4G and 5G in the wild in a metropolis. We separate the traces into 4G and 5G according to the labels.

Metrics. We use the following metrics for evaluation.

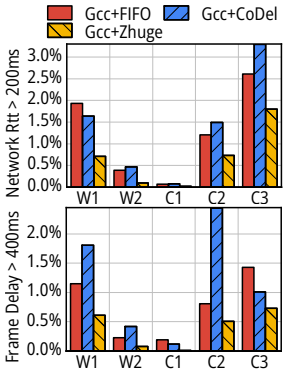


Figure 11: Results of trace-driven simulations over RTP/RTCP.

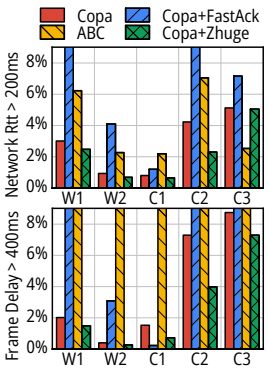


Figure 12: Results of trace-driven simulations over TCP.

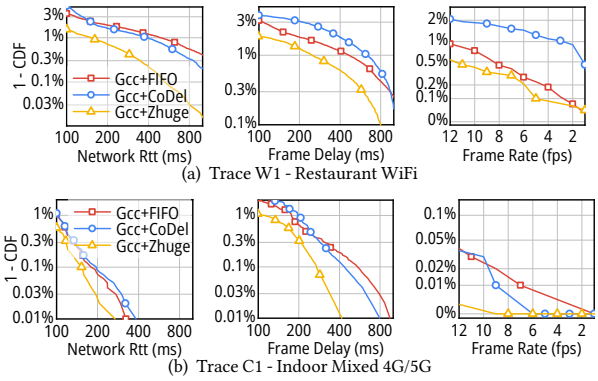


Figure 13: Delay distributions of Zhuge and different baselines over RTP/RTCP. Note that all y-axes are log-scaled.

- *RTT.* We measure the RTT of packets at the network layer. We consider the ratio of RTT >200ms as tail latency ratio.
- *Frame delay.* Frame delay is defined as the time interval between frame encoding at the sender and decoding at the receiver. One frame can only be decoded until all packets of this frame have arrived and previously referred frames have already been decoded. Therefore, frame delay is a direct metric to evaluate latency-related user experience of videos. We consider a frame with delay of >400ms as a delayed frame.
- *Frame rate.* Users will also experience stutters if the frame-rate arriving at the client is too low. Thus, we can also assess video quality according to the frame rate. We consider a per-second frame rate of <10fps as low frame rate.

In this paper, we do not adopt the video quality metrics such as PSNR [4], SSIM [59], and VMAF [44] since they do not reflect the end-to-end interactive delay. Some recent efforts are focused on subjective experience metrics [20], which is left for our future work.

7.3 Trace-driven Simulation

We use NS-3 for simulation to evaluate the tail network latency and application performance of Zhuge under real-world wireless traces. We emulate the bottleneck link in NS-3 with five traces, and evaluate Zhuge over RTP/RTCP and TCP.

RTP/RTCP. As presented in Figure 11, for RTP/RTCP, Zhuge outperforms all baselines in all traces and achieves consistent low latency. Specifically, Zhuge could reduce the ratio of long network RTT by

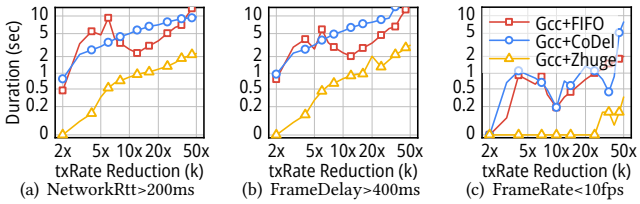


Figure 14: Performance comparison over RTP under ABW drop.

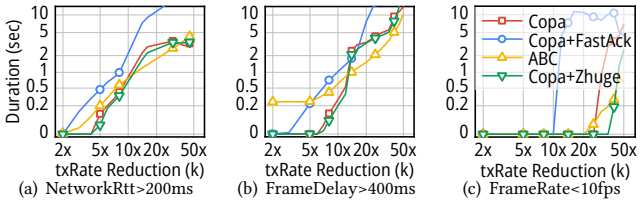


Figure 15: Performance comparison over TCP under ABW drop.

45% to 75% compared with the best baseline. Consequently, the delayed frame ratio is reduced by 38% to 92% in different traces, which significantly reduces video rebuffering and improves user experience. We also observe that Gcc+CoDel outperforms Gcc+FIFO in trace C1 and C3 with respect to frame delay, but falls short in the other three traces. This is because delay-based CCAs like GCC may not be sensitive to packet losses unless it's severe (packet loss rate >10%).

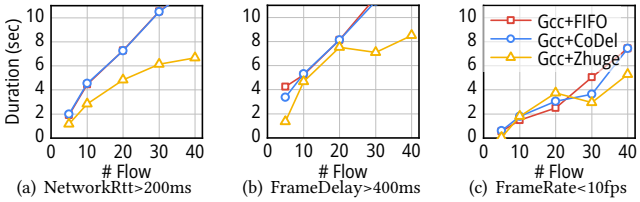
We further present the detailed results of RTP/RTCP based on trace W1 (WiFi) and C1 (cellular) in Figure 13 to better understand the optimization of Zhuge. We observe that Zhuge could reduce the tail latency, long frame delay ratio, and low frame rate ratio at all tail percentiles against two baselines. For example, the P99 tail latency is reduced from 400ms to 170ms, and 400ms delayed frame ratio is reduced from 1% to 0.55% based on trace W1. Moreover, Zhuge could also reduce the ratio of low frame rate by at least 50% in two traces.

TCP. Figure 12 shows that for TCP, as a pure AP-based solution, Zhuge could outperform other AP-based solutions (Copa+FastAck) and achieve comparable performance with end-AP coordinated solution (ABC) in all traces. In terms of tail latency, Copa+Zhuge comprehensively outperforms Copa and Copa+FastAck. We also observe that Copa+FastAck does not consistently perform better than Copa due to FastAck's aggressive retransmission strategy. ABC has a better performance than Copa+Zhuge on trace C3, as ABC could coordinate the AP and end hosts with customized feedback messages, which may not be deployable at scale as discussed in §2.3. For frame delay, Copa+Zhuge achieves the best performance over competitors including ABC in all traces except C1, where Copa+FastAck is slightly better. ABC does not perform well on frame delay due to its aggressive rate ascending design. We further repeat our experiments with the traces used in the ABC paper in Appendix B and find that Zhuge also achieves comparable performance with ABC.

7.4 Microbenchmarks under Wireless Fluctuations

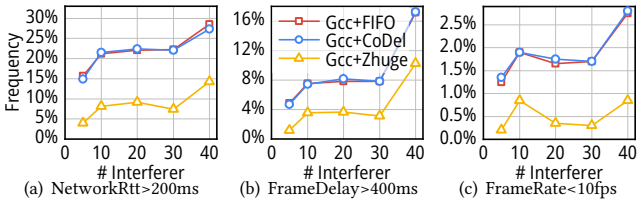
We further simulate the performance of Zhuge under bandwidth reduction, flow competition, and wireless interference.

Bandwidth drop. We evaluate the capability of Zhuge to quickly adapt to bandwidth reduction and reduce the period of network condition and application performance degradation. We first simulate a link with 50ms RTT and 30Mbps bandwidth and start transmission.



(a) NetworkRtt>200ms (b) FrameDelay>400ms (c) FrameRate<10fps

Figure 16: Performance comparison over RTP under competition.



(a) NetworkRtt>200ms (b) FrameDelay>400ms (c) FrameRate<10fps

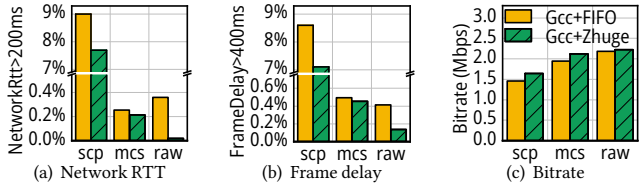
Figure 17: Performance comparison over RTP under interference.

When the CCA reaches the steady state, we reduce the bandwidth by a factor of $k\times$ from 2 to 50, and measure the duration of RTT > 200ms, frame delay > 400ms, and frame rate < 10fps before convergence.

As shown in Figure 14, for RTP/RTCP, Gcc+Zhuge reduces the duration of network degradations and application performance by at least 50% in a wide range of settings. Results over TCP show similar results as presented in Figure 15. Compared with the better performer of Copa and Copa+FastAck, Copa+Zhuge could significantly reduce the duration of high network RTT by 14% to 64.3% when $k < 30$. For $k \geqslant 30$, our observation is that the degradation duration is mainly bounded by the TCP retransmission timeout (RTO) due to severe packet loss, and the performance improvement of Zhuge is not as remarkable. Similarly, Zhuge outperforms ABC when $k < 15$ but under-performs ABC (joint network-host optimization). Nevertheless, according to our measurements in Figure 3(b), 99% bandwidth drop cases fall into $k < 15$, where Zhuge brings good improvements.

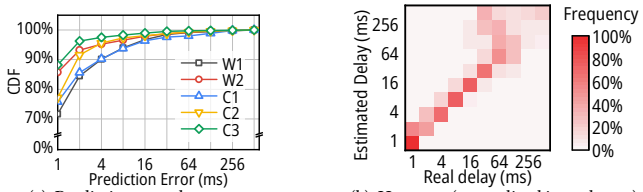
Flow competition. We then investigate how would flows with Zhuge behave when confronting competitors on the same bottleneck queue. We start a different number of bulk transfer flows with TCP CUBIC as competitors and let them compete in the access point. We measure the duration of network RTT >200ms, frame delay >400ms, and frame rate <10fps. Figure 16 shows that compared with FIFO and CoDel, Zhuge could reduce the duration of performance degradation by up to 40% in all cases. Thus, Zhuge could effectively ameliorate the performance degradation under competition.

Wireless interference. We measure the duration of performance degradation with different numbers of wireless interferers. These interferers are also bulk transfer applications based on TCP CUBIC, yet connected to different access points. They compete for the same wireless channel with the RTC flow optimized by Zhuge. We vary the number of interferers from 5 to 40. Note that in the scenario of wireless interference, the interference in wireless channels happens all the time, thus we cannot calculate the degradation duration for a single event as in previous two scenarios. As shown in Figure 17, Zhuge could reduce the frequency of degradation of both network condition and application performance by at least 50%. Note that according to a recent measurement by Cisco [15], there could be up to 29 interferers at the 90th percentile on a 2.4GHz channel. Therefore, Zhuge could bring benefits in a noisy wireless environment.



(a) Network RTT (b) Frame delay (c) Bitrate

Figure 18: Testbed experiments of Zhuge with an RTC flow.



(a) Prediction error by trace. (b) Heatmap (normalized in each row).

Figure 19: Prediction accuracy of Zhuge Fortune Teller.

7.5 Real-World Experiments

We further evaluate the performance of Zhuge with our OpenWrt-based WiFi AP testbed. We set up an RTC server and a client with the WebRTC APIs [9] in Microsoft Edge browsers on two laptops. The server streams a timestamped video to the client through the `peerconnection` API over RTP/RTCP and GCC. The server is wire-connected to the AP, while the client connects to AP through WiFi. We evaluate the performance of Zhuge in the following scenarios, each lasting for 6 hours.

- `scp`. This experiment is designed to evaluate the performance of Zhuge over RTC flows when competing with other flows. We periodically start and stop an scp file transmission from the server to the client every 30 seconds.
- `mcs`. This experiment is designed to mimic fluctuating wireless channels. 802.11 access points will dynamically change the modulation coding scheme (MCS) at the link layer to adapt to channel conditions. Therefore, similar to [31], we randomly change the MCS every 30 seconds with the Linux `iw` command and assess Zhuge's reaction to fluctuation.
- `raw`. We report the results of running the RTC application in our crowded office without additional configurations.

We measure the network RTT by analyzing the packet captures, and frame delay by calculating the timestamp difference between video sent and video received. As shown in Figure 18(a) and 18(b), both the network RTT and frame delay of the RTC flow with Zhuge has been improved against baselines by 17% to 95% (network RTT) and 9% to 67% (frame delay) in all scenarios. This indicates that Zhuge could effectively reduce the tail latency in real wireless environments.

Meanwhile, we also evaluate the capability of Zhuge to maintain similar performance in a steady wireless channel compared with the baseline. We evaluate the steady-state performance by measuring the video's average bitrate based on Microsoft Edge and present the results in Figure 18(c). We observe that Zhuge could maintain similar average bitrate, demonstrating its maintenance of performance in the steady state. Note that the improvement in tail latency is not reflected in the bitrate results.

7.6 Zhuge Deep Dive

Finally, we report the fairness and runtime overhead of Zhuge.

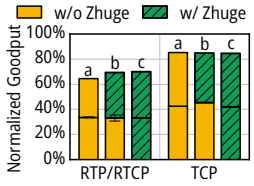


Figure 20: Fairness of Zhuge.

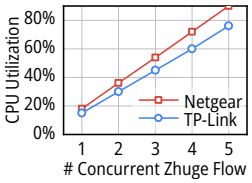


Figure 21: CPU Overhead.

Estimation accuracy. We measure the accuracy in the estimation of packet delay in §4.1. We compare the estimated delay and the real delay measured later for the same packet. We present the distribution of the prediction error in different traces in Figure 19(a). In most cases, the prediction error is much less than the RTT in our experiment (50ms). We also put the different results into bins and present the heatmap of the frequency of each bin. As shown in Figure 19(b), when the estimated delay is low (1-64ms), the estimation is usually accurate. When the estimated delay is high (>64ms), the estimation could be inaccurate, but the real delays are still high enough (more than one RTT) to trigger the sender to reduce the sending rate.

Internal fairness. We analyze whether Zhuge affects the bitrate fairness in the steady state when optimizing two RTC flows simultaneously. We report the goodput of RTC flows normalized by the link capacity when they compete for the same AP. Bar *a* in Figure 20 reports the goodput of two flows without Zhuge, while Bar *c* reports the goodput when both flows are optimized by Zhuge. We discover that the bitrate fairness in the steady state is not affected by Zhuge with GCC over RTP/RTCP or Copa over TCP. For GCC, Zhuge even slightly increases the average flow bitrate by 10%. This is because Zhuge enables the sender to react faster to the situation where the sending rate oversteps the link capacity.

External fairness. We evaluate whether Zhuge advantages optimized flows by compromising other flows with the same CCAs during competition. We measure the bitrate of two RTC flows, one of which is optimized by Zhuge and the other one is not. We present the results in the bar b in Figure 20. For both GCC and Copa, the bitrate difference of the two flows are < 3%. Thus, as discussed in §6, the performance improvement of Zhuge is not built on sacrificing the performance of other flows. Instead, two flows compete fairly, as intended by CCAs.

CPU overhead. We measure the CPU utilization of Zhuge with our implementation on an OpenWrt-based Netgear WiFi AP, as well as a TP-Link TL-WDR4900 [2] AP. We measure the CPU utilization when processing different numbers of concurrent unencrypted RTC flows by Zhuge, and present the result in Figure 21. These two APs manufactured ten years ago could still support Zhuge to process 5 concurrent RTC flows, which can cover many real scenarios (e.g., home WiFi).

There are several potential directions to optimize the resource overhead of Zhuge. First, when the CPU utilization is high, instead of estimating all the downlink packets, Zhuge could selectively update the network conditions. As long as the time interval between estimation is negligible (e.g., several milliseconds), the control loop is still reduced. Moreover, our prototype implementation of Zhuge is based on user-space packet sockets, which could be further optimized by inserting Zhuge as a kernel module. Finally, there are also successful deployment of other per-packet state maintenance features in commercial APs [15, 43].

8 Related Work

Wireless performance optimization. Besides solutions discussed in §2.3, there are also several related research efforts in the network layer and above to improve the performance of wireless networks. One line of solutions is to design specific CCAs such as Sprout [60], Verus [64], and others. Zhuge by design could work together with these algorithms, similar to working with Copa in §7. There are also proposals to decouple the wired connection from the wireless connection [16] or to better manage the retransmission from routers [13, 15]. The proxy independently manages the sending rate on both links to reduce the control loop for the transport layer. However, for RTC applications, contents are generated from the sender (e.g., encoder). Therefore, the control loop is still high in terms of the application layer. Beyond solutions purely in the network layer and above, there are also many research and industry efforts to improve the wireless performance by cross-layer designs [53, 61]. These solutions need considerable integration with the server or the client, which might prevent their deployment at scale.

Transient performance. As discussed in §2.3, with the development of steady-state performance of CCAs, the transient performance is attracting more and more attention. In the research of the transport layer, there are also recent interests in the analysis or quantification of the transient performance of CCAs in both WAN [57, 63] and data centers [45]. These solutions still address or analyze the transient performance from the view of the server. In contrast, Zhuge presents a deployable solution to improve transient performance from the vantage point of access points in wireless networks.

Transport optimizations for RTC. There are also several transport innovations to achieve consistent performance for the RTC applications. For example, proposals in [19, 27] optimize the retransmission mechanism of RTC applications. Fouladi et al. [28] further adapts the stream encoder to network conditions in order to reduce degradation of the application performance (e.g., frame delay). However, the bandwidth drops due to external factors are not predictable in advance. In contrast, Zhuge focuses on the timely reaction to network conditions, which could also work together with these research efforts. We leave the joint optimization with other control mechanisms beyond CCA for future work.

9 Conclusion

We propose Zhuge, an in-AP solution that reduces the control loop to alleviate tail latency for RTC applications in wireless networks. Zhuge predicts the fortune of each packet upon its arrival with the Fortune Teller, and quickly notify the sender about these fortunes over a variety of protocols with the Feedback Updater. We evaluate the performance of Zhuge with both real-world trace-driven simulations and deployments in the testbed. Experiments show that Zhuge reduces the tail of long latency and RTC application performance degradation by 17% to 95% in different scenarios.

This work does not raise any ethical issues.

Acknowledgements. We sincerely thank our shepherd Brad Karp, anonymous reviewers, and labmates in Routing Group from Tsinghua University for their valuable feedback. This work is sponsored by National Natural Science Foundation of China (No.62002196 and 61832013), Alibaba Innovative Research (AIR) Program, and National Science Foundation (No. 1850384). Bo Wang and Chen Sun are the corresponding authors.

References

[1] 2011. [OpenWrt Wiki] NETGEAR WNDR3800. https://openwrt.org/toh/netgear/wndr3800.

[2] 2013. [OpenWrt Wiki] TP-Link TL-WDR4900. https://openwrt.org/toh/tp-link/tl-wdr4900.

[3] 2014. [systemd-devel] [ANNOUNCE] systemd 217. https://lists.freedesktop.org/archives/systemd-devel/2014-October/024662.html.

[4] 2020. Peak signal-to-noise ratio - Wikipedia. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio.

[5] 2020. Start - Tencent Cloud Gaming. https://start.qq.com/.

[6] 2021. Manwuyixiang Roast Lamb Leg 【满屋溢香烤羊腿·烤海鲜(双清路店)】. http://cnc.www.dianping.com/shop/igEL946mgXy0B2KV.

[7] 2021. Prepare your network for Meet video calls - Google Workspace Admin Help. https://support.google.com/a/answer/1279090.

[8] 2021. Troubleshooting your Stadia experience - Stadia Help. https://support.google.com/stadia/answer/9595943.

[9] 2021. WebRTC Samples. https://webrtc.github.io/samples/.

[10] 2021. Zoom network firewall or proxy server settings – Zoom Support. https://support.zoom.us/hc/en-us/articles/201362683-Zoom-network-firewall-or-proxy-server-settings.

[11] Soheil Abbasloo, Chen-Yu Yen, and H Jonathan Chao. 2020. Classic meets modern: A pragmatic learning-based congestion control for the Internet. In *Proc. ACM SIGCOMM*.

[12] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *Proc. USENIX NSDI*.

[13] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H Katz. 1995. Improving TCP/IP performance over wireless networks. In *Proc. ACM MobiCom*.

[14] Mark Baugher, D McGrew, M Naslund, E Carrara, and Karl Norrman. 2004. The secure real-time transport protocol (SRTP). *IETF RFC 3711* (2004).

[15] Apurv Bhartia, Bo Chen, Feng Wang, Derrick Pallas, Raluca Musaloiu-E, Ted Tsung-Te Lai, and Hao Ma. 2017. Measurement-based, practical techniques to improve 802.11 ac performance. In *Proc. ACM IMC*.

[16] John Border, Markku Kojo, Jim Griner, Gabriel Montenegro, and Zach Shelby. 2001. Performance enhancing proxies intended to mitigate link-related degradations. *IETF RFC 3135* (2001).

[17] brianhu. 2021. Google Meet Troubleshooting Playbook - Network and Hardware Troubleshooting. https://www.googlecloudcommunity.com/gc/Workspace-Product-Articles/Google-Meet-Troubleshooting-Playbook-Network-and-Hardware/ta-p/165810.

[18] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. *ACM Queue* (2016).

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

[20] Ke Chen, Han Wang, Shuwen Fang, Xiaotian Li, Minghao Ye, and H. Jonathan Chao. 2022. RL-AFEC: Adaptive Forward Error Correction for Real-time Video Communication Based on Reinforcement Learning. In *Proc. ACM MMSys*.

[21] Yusuf Cinar, Peter Pocta, Desmond Chambers, and Hugh Melvin. 2021. Improved jitter buffer management for WebRTC. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* (2021).

[22] Yousri Daldoul, Djamal-Eddine Meddour, and Adlen Ksentini. 2020. Performance Evaluation of OFDMA and MU-MIMO in 802.11 ax Networks. *Computer Networks* (2020).

[23] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. 2021. A network analysis on cloud gaming: Stadia, GeForce Now and PSNow. *Network* (2021).

[24] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC vivace: Online-learning congestion control. In *Proc. USENIX NSDI*.

[25] Theodore Faber. 1998. ACC: using active networking to enhance feedback congestion control mechanisms. *IEEE network* (1998).

[26] Marcel Flores, Alexander Wenzel, and Aleksandar Kuzmanovic. 2016. Enabling router-assisted congestion control on the Internet. In *Proc. IEEE ICNP*.

[27] Silas L Fong, Salma Emara, Baochun Li, Ashish Khisti, Wai-Tian Tan, Xiaoqing Zhu, and John Apostolopoulos. 2019. Low-latency network-adaptive error control for interactive streaming. In *Proc. ACM Multimedia*.

[28] 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 *Proc. USENIX NSDI*.

[29] Nitin Garg. 2019. COPA congestion control for video performance - Engineering at Meta. https://engineering.fb.com/2019/11/17/video-engineering/copa/.

[30] Moinak Ghoshal, Pranab Dash, Zhaoning Kong, Qian Xu, Y.Charlie Hu, Dimitrios Koutsonikolas, and Yuanjie Li. 2022. Can 5G mmWave Enable multi-user AR apps?. In *Proc. PAM*.

[31] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. 2020. ABC: A Simple Explicit Congestion Controller for Wireless Networks. In *Proc. USENIX NSDI*.

[32] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review* (2008).

[33] Toke Høiland-Jørgensen, Michał Kazior, Dave Täht, Per Hurtig, and Anna Brunstrom. 2017. Ending the anomaly: Achieving low latency and airtime fairness in wifi. In *Proc. USENIX ATC*.

[34] Toke Høiland-Jørgensen, Dave Täht, and Jonathan Morton. 2018. Piece of CAKE: a comprehensive queue management solution for home gateways. In *Proc. IEEE LANMAN*.

[35] Stefan Holmer, Magnus Flodman, and Erik Sprang. 2015. RTP extensions for transport-wide congestion control. https://datatracker.ietf.org/doc/html/draft-holmer-rmcat-transport-wide-cc-extensions-01.

[36] Jana Iyengar and Ian Swett. 2021. Quic loss detection and congestion control. *IETF RFC 9002* (2021).

[37] Ingemar Johansson and Zaheduzzaman Sarker. 2017. Self-Clocked Rate Adaptation for Multimedia. IETF RFC 8298.

[38] Alan Jones, Peter Sevcik, and Rebecca Wetzel. 2021. Internet Connection Requirements for Effective Video Conferencing to Support Work from Home and eLearning | NetForecast. https://www.netforecast.com/wp-content/uploads/NFR5137-Videoconferencing_Internet_Requirements.pdf.

[39] Teemu Kämäräinen, Matti Siekkinen, Antti Ylä-Jääski, Wenxiao Zhang, and Pan Hui. 2017. A measurement study on achieving imperceptible latency in mobile cloud gaming. In *Proc. ACM MMSys*.

[40] Ad Kamerman and Leo Monteban. 1997. WaveLAN®-II: a high-performance wireless LAN for the unlicensed band. *Bell Labs technical journal* (1997).

[41] Dina Katabi, Mark Handley, and Charlie Rohrs. 2002. Congestion control for high bandwidth-delay product networks. In *Proc. ACM SIGCOMM*.

[42] Erik Kjerland, Matt Shadbolt, Anthony Watherston, Alma Jenks, and Doug Eby. 2021. Network requirements for Windows 365 | Microsoft Docs. https://docs.microsoft.com/en-us/windows-365/enterprise/requirements-network.

[43] Ingo Kofler, Martin Prangl, Robert Kuschnig, and Hermann Hellwagner. 2008. An H. 264/SVC-based adaptation proxy on a WiFi router. In *Proc. NOSSDAV*.

[44] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. 2016. Toward A Practical Perceptual Video Quality Metric | Netflix TechBlog. https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652.

[45] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. 2021. Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport. In *Proc. USENIX NSDI*.

[46] Jason Livingood. 2021. Working latency — the next QoE frontier | APNIC Blog. https://blog.apnic.net/2021/12/02/working-latency-the-next-qoe-frontier/.

[47] Bill Marczak and John Scott-Railton. 2020. Move Fast and Roll Your Own Crypto: A Quick Look at the Confidentiality of Zoom Meetings - The Citizen Lab. https://citizenlab.ca/2020/04/move-fast-roll-your-own-crypto-a-quick-look-at-the-confidentiality-of-zoom-meetings/.

[48] Gustavo Marfia, Claudio E Palazzi, Giovanni Pau, Mario Gerla, and Marco Roccetti. 2010. TCP Libra: Derivation, analysis, and comparison with other RTT-fair TCPs. *Computer Networks* (2010).

[49] Zili Meng, Yaning Guo, Yixin Shen, Jing Chen, Chao Zhou, Minhu Wang, Jia Zhang, Mingwei Xu, Chen Sun, and Hongxin Hu. 2021. Practically Deploying Heavyweight Adaptive Bitrate Algorithms With Teacher-Student Learning. *IEEE/ACM Transactions on Networking* (2021).

[50] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. 2020. Pruning Edge Research with Latency Shears. In *Proc. ACM HotNets*.

[51] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. 2020. A first look at commercial 5G performance on smartphones. In *Proc. WWW*.

[52] Kathleen Nichols and Van Jacobson. 2012. Controlling queue delay. *Commun. ACM* (2012).

[53] Shinik Park, Jinsung Lee, Junseon Kim, Jihoon Lee, Sangtae Ha, and Kyunghan Lee. 2018. ExLL: An extremely low-latency congestion control for mobile cellular networks. In *Proc. ACM CoNEXT*.

[54] Carolyn Rowe, Diana Hanson, Chiffers Craig, David Coulter, Justin Gilmore, David Byrd, Ajayan Borys, Kelly Baker, Baard Hermansen, Serdar Soysal, et al. 2021. Microsoft Teams call flows - Microsoft Teams | Microsoft Docs. https://docs.microsoft.com/en-us/microsoftteams/microsoft-teams-online-call-flows.

[55] Zaheduzzaman Sarker, Colin Perkins, Varun Singh, and M Ramalho. 2021. RTP Control Protocol (RTCP) Feedback for Congestion Control. *IETF RFC 8888* (2021).

[56] Yueshi Shen. 2017. Live Video Transmuxing/Transcoding: FFmpeg vs Twitch-Transcoder, Part I | Twitch Blog. https://blog.twitch.tv/en/2017/10/10/live-video-transmuxing-transcoding-f-fmpeg-vs-twitch-transcoder-part-i-489c1c125f28/.

[57] Yixin Shen, Zili Meng, Jing Chen, and Mingwei Xu. 2021. Quantifying the transient performance of congestion control algorithms. In *Proc. ACM SIGCOMM Poster and Demo*.

[58] C-H Tai, Jiang Zhu, and Nandita Dukkipati. 2008. Making large scale deployment of RCP practical for real networks. In *Proc. IEEE INFOCOM*.

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

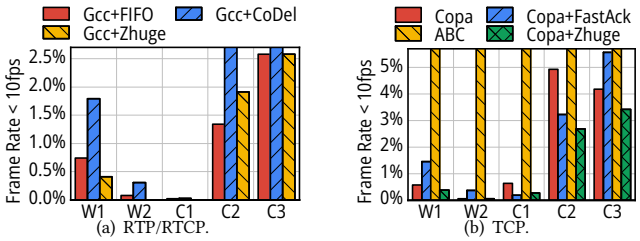


Figure 22: The ratio of frame rate<10fps over real-world traces.

on Image Processing (2004).

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

[61] Yaxiong Xie, Fan Yi, and Kyle Jamieson. 2020. PBE-CC: Congestion control via endpoint-centric, physical-layer bandwidth measurements. In *Proc. ACM SIGCOMM.*

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

[63] Yang Richard Yang, Nin Sik Kim, and Simon S Lam. 2001. Transient behaviors of TCP-friendly congestion control protocols. In *Proc. IEEE INFOCOM.*

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

[65] Xiaoqing Zhu, Rong Pan, Michael A. Ramalho, and Sergio Mena de la Cruz. 2020. Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media. IETF RFC 8698.

Appendices

Appendices are supporting material that has not been peer-reviewed.

A Measurement Details

We carried out two measurements in this paper, including the measurement of the network conditions and application performance of our online RTC application in §2.3, and the trace collection of available bandwidth from WiFi networks in §7.2. We present their measurement details as below.

Performance of our online RTC application. We measure our online RTC application for one month in December 2021, with millions of user sessions, and billions of video frames. Among them, the Ethernet, WiFi, and 4G are the top-three types of access networks in our users. We then calculate the tail performance metrics as shown in §2.3.

Available bandwidth of WiFi networks. We measure the available bandwidth of the WiFi network in a nearby restaurant [6], and in our office. We continuously download a large file from another Ethernet-connected server in the same subnet with `wget`. To bypass the potential rate limits over the UDP protocol, we run TCP CUBIC on the server. We calculate the receiving rate from the packet captures at the client as the available bandwidth. The average receiving rate of the office WiFi and restaurant WiFi are 27Mbps and 21Mbps respectively.

B Supplementary Trace-Driven Simulations

Frame-rate improvements. We further present the summary of the performance improvements on the frame-rate in Figure 22. We measure the ratio of low frame-rate (per-second frame rate <10fps). As shown in Figure 22(a) and 22(b), Zhuge achieves the smallest (or close to smallest) low frame rate ratio among all baselines. ABC does not perform well in terms of frame rate in these five traces due to its aggressiveness on rate increasing, which we will further analyze below.

	Copa	ABC	Copa+Zhuge
P(NetworkRtt > 200ms)	0.1%	6.4%	0.1%
P(FrameDelay > 400ms)	9.5%	2.4%	3.2%
P(FrameRate < 10fps)	4.5%	0.8%	1.5%

Table 3: Performance of on the original traces of ABC.

Results over the traces used in ABC [31]. We further rerun the simulation over the original traces evaluated in the ABC paper. We find that ABC does perform the best among all solutions in terms of application performance (frame delay and frame rate). Nevertheless, Zhuge could still significantly improve the application performance against the original Copa by 67% and achieve comparable performance to ABC. This indicates that Zhuge could achieve comparable performance without modifications on the server or the client like ABC. We do not present this result in the main text since the traces evaluated in ABC were collected 10 years ago while other traces are collected in recent 2 years. The average available bandwidth of ABC traces is an order of magnitude lower than that in the 5 traces in §7.2. Thus, the traces in ABC may not faithfully reflect the development of the wireless access networks in recent years.