Latency Optimization in Real-Time Multimedia Streaming

Translated from Chinese
in October 2023

Zili Meng

a dissertation
presented to the faculty
of Tsinghua University
in candidacy for the degree
of Doctor of Philosophy

recommended for acceptance
by the Institute for
Network Sciences and Cyberspace

Adviser: Mingwei Xu

June 2023

Abstract

Real-time multimedia streaming is one of the most important applications in the Internet, and has a stringent requirement for latency. Existing solutions cannot fully satisfy the requirements of real-time multimedia streaming applications in many parts of the Internet architecture. Among them, latency fluctuation is the most challenging problem in the latency optimization. This dissertation focuses on the latency issue of real-time multimedia streaming and systematically optimizes the latency fluctuation thoroughly from many aspects of the Internet architecture. The main contributions of this dissertation are as follows:

1. To address the issue of heterogeneous latency contributors of real-time multimedia transport, this dissertation proposes the architecture of real-time multimedia transport. It identifies that the heterogeneous latency contributors of real-time multimedia transport are mainly caused by the control path and data path of the system. This dissertation further analyzes how control path and data path affect the latency of real-time multimedia transport, and optimizes each part respectively.

2. To address the latency fluctuations on the control path, this dissertation separates the control path into feedback and decision-making, and proposes Zhuge and Metis, respectively, to control the performance fluctuation of these two parts. Zhuge identifies that the inflation of feedback delay prevents the sender from adjusting the sending rate of multimedia in time. Zhuge proposes a feedback mechanism that separates the control path and data path to improve performance. Metis notes that the increasing complexity of the decision algorithm may cause delayed or erroneous decisions, which leads to performance fluctuations. Metis converts complex algorithms into low-latency and interpretable decision trees to mitigate performance fluctuations. Experimental results based on real-world traffic show that up to 75% latency fluctuations can therefore be mitigated.

3. To address the latency fluctuations on the data path, this dissertation proposes AFR, Hairpin and Confucius, respectively, to control the latency fluctuations of the application layer, transport layer and network layer. AFR addresses the issue of application-level delay fluctuations caused by the delay of the application decoder. AFR proposes an adaptive frame-rate management mechanism to reduce the delay fluctuations of the application layer. Hairpin addresses the issue that the existing loss recovery schemes cannot meet the requirements of real-time multimedia streaming applications due to the stringent requirements of latency fluctuations. Hairpin proposes a joint loss recovery scheme that combines

retransmission and redundant recovery to control the delay fluctuations caused by packet loss. Confucius addresses the issue that the delay fluctuations caused by unknown competing flows and interference on the network layer cannot be controlled by the existing fairness mechanisms. Confucius proposes a progressive active queue management mechanism to control the delay fluctuations on the network layer while ensuring fairness. Experimental results show that AFR, Hairpin and Confucius can reduce the latency fluctuations of real-time multimedia streaming by 13-67% in different scenarios.

Contents

Listing of figures

List of Tables

Acknowledgments

Acknowledgment is usually the first part I would like to read when reading a dissertation – it includes all the emotions, feelings, ups and downs in the life. The Chinese version of acknowledgment has a one-page limit, so I am rewriting another acknowledgment here.

Looking back to 2016, the first step into Room 4-204 in the FIT Building was my starting point for working in computer networks. By then, I had no idea what the future would be like, only knowing that it was yet another research experience – I had one unsuccessful trial in other directions before and was hoping that computer networks would excite me. Looking back, it was a splendid trip with all my dear labmates, friends, supervisors, and collaborators. I am hugely grateful to all the people I met all the way. In the office (Room 4-204) and machine room (Room 3-229), we cried, laughed, commiserated rejections, and cheered successes together. That is indeed an unforgettable experience in my life.

First, I would definitely thank my advisor, Prof. Mingwei Xu. Besides the research side, I'd like to thank Mingwei more in the way that Mingwei is always the one that you can rely on when you get into trouble outside. I still remember the first day of meeting with Mingwei. We discussed several projects, and in the end, he asked "are you from Liaoning (my hometown)?" – I have a strong Mandarin accent, and Mingwei can sharply recognize it. Throughout my whole PhD life, I have to say I'm more like a troublemaker – on student affairs, on industrial collaborations, and many other aspects. Mingwei always stands with me, supporting me to explore whatever I want.

I am also grateful to my undergraduate thesis advisor, Prof. Jun Bi. I started to work with Jun in 2016, and I was always impressed by the passion of Jun. I would say that significantly shaped my personality – stay determined and always believe a bright future will happen. I still remember the talk with Jun in 2018 – we talked a lot about being as broad as a career path and as specific as the ongoing research project. Finally, I made up my mind to work with Jun for PhD. Unfortunately, he passed away in 2019, but his spirit will definitely be carried.

I'd also like to say a big thank you to Prof. Justine Sherry. I wrote a cold email to Justine in Sep 2021, asking if I could work with her for several months. Later, in a Zoom chat, I clearly felt the enthusiasm for research and excitement about network problems. Justine's

group is the first group I feel like a family when working in the group. The selfless help from Justine always moved me and motivated me when I felt blue. I would say without Justine, I could not imagine working as a faculty in a prestigious university now. I have worked with Justine till now and am still learning from her wisdom on being a faculty.

I would also like to thank my (co-)advisors during my different periods of experience. Dr. Chen Sun patiently taught me how to get hands-on from a rough idea to a solid paper. I worked with him as an undergraduate at Tsinghua and as an intern at Alibaba, both of which benefited me a lot! Prof. Mohammad Alizadeh from Massachusetts Institute of Technology hosted and advised me in the summer of 2018. The unbelievably smart brain from him lets me see what is the top-tier intelligence in the world. Prof. Hongxin Hu from University at Buffalo has advised me since 2017. I am very fortunate to have learned a lot about the foreseeing ability of research problems. Mr. Rui Han was my mentor during my internship at Tencent. He has a respectful character, a humble outlook, and a very solid programming foundation. I clearly learned a lot from his extraordinary system engineering ability from him. Dr. Hongzi Mao from Massachusetts Institute of Technology mentored me during my visit to Mohammad's group – which significantly opened my eyes to doing research from a totally different ML-oriented mindset. I hope he will feel happy that I'm working at his alma mater. Thanks also go to my leaders during the internship – Dr. Harry Liu from Alibaba, Mr. Wei Yang from Tencent, and Mr. Zhaosong Ruan from JLSemi. My internships cannot go smooth without your generous support.

I would also like to thank all the collaborators from Anmin Xu, Shuhe Wang, Minhu Wang, Haiping Wang, Jiasong Bai (Tsinghua University), Venkat Arun (MIT), Tong Yang (Peking University), Qun Huang (Chinese Academy of Sciences), Nirav Atre, Maria Apostolaki (Carnegie Mellon University), Xue Wei (Tencent), and Chao Zhou (Kuaishou). Without your support, there is no way to finish the projects in this thesis. Thanks to numerous collaborators for the projects that I participated in as well.

I'm also extremely fortunate to work with all genius labmates. Needless to say, many of them are now professors in different places. I want to thank Jia Zhang for her patience when I contacted my supervisor, her help when I first joined the group, and her care when I encountered difficulties in life. I want to thank Jiang Li for every morning when we played badminton together and for every moment when we went traveling, hiking, and dining out. Without you, I wouldn't be where I am today. Thanks to Bingyang Liu, Yu Zhou, Menghao Zhang, Cheng Zhang, Kai Gao, Zhilong Zheng, Yunsenxiao Lin, Dai Zhang, Heng Yu, Guanyu Li, Jiamin Cao, and so many other labmates for the support all the time. Also thanks to Miguel Ferreira, Hugo Sadok, Adithya Philip, Margarida Ferreira, Chris Canel, Francisco Pereira, and all my friends when I visited CMU.

I would also like to thank Prof. Bo Wang, Prof. Enhuan Dong, Xiao, Tingfeng, Jing, Yixuan, Yaning, Yixin, Yuxi, and everyone working with me. It is collaborating with you

guys that teach me how to participate in a project. Also, thank you for your tolerance and inclusiveness for me all the time.

Thank you to my family. Thanks to my parents for their unwavering support and trust in me all the way, allowing me to have the confidence to do whatever I want. You always support my decisions in each critical moment in my life. Thanks to my roommates (Xu Shi, Zhenyu Qin, Shuo Zhang, Xiang Li, and Juncai Liu), you have always been my support during my ups and downs.

I finally want to thank the era that I am in. When I started to work on video streaming back in 2018, it was even before the pandemic – I guess no one would expect that real-time video streaming would be a part of our lives. My achievements are mostly attributed to the opportunities provided in this special era, which enables me to contribute to improving people's living quality. And we are also expecting an even more advanced era driven by real-time video streaming in the future.

I heard running rumors saying that the new building for computer science will be put in use from this December, so Room 3-229 and 4-204 might be the home that I can never go back to. Let's call it an experience in our deep minds.

Oct 2023
Hong Kong University of Science and Technology

Unfortunately there is no window in my office. But after advocating sending rates taking over congestion windows for years, it might be the right place for me :-)

Previously Published Material

Chapter 3 revises a previous publication [184]: Meng, Z., Xu, M. Latency Optimization in Real-Time Multimedia Transport: Architecture, Progress and the Future. *Journal of Computer Research and Development*, 2023 (in Chinese).

Chapter 4 revises a previous publication [188]: Meng, Z., Guo, Y., Sun, C., Wang, B., Sherry, J., Liu, H. H., Xu, M. Achieving Consistent Low Latency for Wireless Real Time Communications with the Shortest Control Loop. In *Proc. ACM SIGCOMM*, 2022.

Chapter 5 revises a previous publication [186]: Meng, Z., Wang, M., Bai, J., Xu, M., Mao, H., Hu, H. Interpreting deep learning-based networking systems. In *Proc. ACM SIGCOMM*, 2020.

Chapter 6 revises a previous publication [190]: Meng, Z., Wang, T., Shen, Y., Wang, B., Xu, M., Han, R., Liu, H., Arun, V., Hu, H., Wei, X. Enabling high quality real-time communications with adaptive frame-rate. In *Proc. USENIX NSDI*, 2023.

Chapter 7 revises a previous publication [191]: Meng, Z., Kong, X., Chen, J., Wang, B., Xu, M., Han, R., Liu, H., Arun, V., Hu, H., and Wei, X. Hairpin: Rethinking packet loss recovery in edge-based interactive video streaming. In *Proc. USENIX NSDI*, 2024.

Chapter 8 revises a previous preprint [189]: Meng, Z., Atre, N., Xu, M., Sherry, J., Apostolaki, M. Confucius Queue Management: Be Fair but not Too Fast. In *arXiv Preprint 2310.18030*, 2023.

Biographical Sketch

Personal Experiences

Aug 2015 – Jul 2019	B.Eng., Department of Electronic Engineering, Tsinghua University, Beijing, China.
Jun 2018 – Sep 2018	Research Assistant, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.
Aug 2019 – Jun 2023	Ph.D., Institute for Network Science and Cyberspace, Tsinghua University, Beijing, China.
Jun 2020 – Sep 2020	Research Intern, Tencent, Shenzhen, China
Jun 2021 – Aug 2021	Research Intern, JLSemi, Nanjing, China
Aug 2021 – Jan 2022	Research Intern, Alibaba, Beijing, China
Feb 2022 – Dec 2022	Research Assistant, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.

Awards

2023 ACM SIGCOMM China Dissertation Award
Outstanding Ph.D. Thesis Award at Tsinghua University

2022 ByteDance Scholar

2021 Best Paper Award, IEEE/ACM IWQoS 2021

2020 Best Paper Award, IEEE ICC 2020
Microsoft Research PhD Fellowship (Asia)

2019 Outstanding B.Eng. Award at Tsinghua University

2018 ACM SIGCOMM 2018 Student Research Competition Gold Medal
Tsinghua Top Grade Scholarship

Publications

[1] **Zili Meng**, X. Kong, J. Chen, B. Wang, M. Xu, et al. "Hairpin: Rethinking Packet Loss Recovery in Edge-based Interactive Video Streaming". In *USENIX Symposium on Networked Systems Design and Implementation* (**NSDI '24**).

[2] **Zili Meng**, T. Wang, Y. Shen, B. Wang, M. Xu, et al. "Enabling High Quality Real-Time Communications with Adaptive Frame-Rate". In *USENIX Symposium on Networked Systems Design and Implementation* (**NSDI '23**).

[3] **Zili Meng**, Y. Guo, C. Sun, B. Wang, J. Sherry, et al. "Achieving Consistent Low Latency for Wireless Real-Time Communications with the Shortest Control Loop". In *ACM SIGCOMM Conference* (**SIGCOMM '22**).

[4] **Zili Meng**, Y. Guo, Y. Shen, J. Chen, C. Zhou, et al. "Practically Deploying Heavyweight Adaptive Bitrate Algorithms With Teacher-Student Learning", *IEEE/ACM Transactions on Networkings* (**ToN**), 2021.

[5] **Zili Meng**, M. Wang, J. Bai, M. Xu, H. Mao, et al. "Interpreting Deep Learning-Based Networking Systems". In *ACM SIGCOMM Conference* (**SIGCOMM '20**).

[6] **Zili Meng**, J. Chen, Y. Guo, C. Sun, H. Hu, et al. "PiTree: Practically Implementing ABR Algorithms Using Decision Trees". In *ACM International Conference on Multimedia* (**MM '19**).

[7] **Zili Meng**, J. Bi, H. Wang, C. Sun, H. Hu. "MicroNF: An Efficient Framework for Enabling Modularized Service Chains in NFV", *IEEE Journal on Selected Areas in Communications* (**JSAC**), 2019.

[8] J. Zhang, S. Ren, E. Dong, **Zili Meng**, Y. Yang, et al. "Reducing Mobile Web Latency through Adaptively Selecting Transport Protocol", *IEEE/ACM Transactions on Networkings* (**ToN**), 2023.

[9] J. Zhang, Y. Zhang, E. Dong, Y. Zhang, S. Ren, **Zili Meng**, et al. "Bridging the Gap between QoE and QoS in Congestion Control: A Large-scale Mobile Web Service Perspective", In *USENIX Annual Technology Conference* (**ATC '23**).

[10] C. Miao, M. Chen, A. Gupta, **Zili Meng**, L. Ye, et al. "Detecting Ephemeral Optical Events with OpTel". In *USENIX Symposium on Networked Systems Design and Implementation* (**NSDI '22**).

[11] J. Zhang, E. Dong, **Zili Meng**, Y. Yang, M. Xu, et al. "WiseTrans: Adaptive Transport Protocol Selection for Mobile Web Service", In *The Web Conference* (**WWW '21**).

[12] X. Chen, Q. Huang, P. Wang, **Zili Meng**, H. Liu, et al. "LightNF: Simplifying Network Function Offloading in Programmable Networks", In *IEEE/ACM International Symposium on Quality of Service* (**IWQoS '21**).

[13] J. Chen, **Zili Meng**, Y. Guo, M. Xu, H. Hu. "HierTopo: Towards High-Performance and Efficient Topology Optimization for Dynamic Networks", In *IEEE/ACM International Symposium on Quality of Service* (**IWQoS '21**).

[14] S. Wang, C. Sun, **Zili Meng**, M. Wang, J. Cao, et al. "Martini: Bridging the Gap between Network Measurement and Control Using Switching ASICs", In *IEEE International Conference on Network Protocols* (**ICNP '20**).

[15] H. Mao, M. Schwarzkopf, S. Venkatakrishnan, **Zili Meng**, M. Alizadeh. "Learning Graph-based Cluster Scheduling Algorithms". In *ACM SIGCOMM Conference* (**SIGCOMM '19**).

Patents

[16] M. Xu, **Zili Meng**, M. Wang, J. Bai. An Interpretation Method for Deep Learning-Based Global Networked Systems. CN111753892A, Chinese Patent. Granted: September 9, 2022.

[17] Z. Ruan, **Zili Meng**, Y. Huang. A Scheduling Method, Device, and Electronic Equipment. CN113872887A, Chinese Patent. Granted: August 16, 2022.

[18] M. Xu, J. Zhang, E. Dong, **Zili Meng**, Y. Yang. Adaptive Transport Protocol Selection Method and Device for Mobile Web Service. CN112583818A, Chinese Patent. Granted: December 24, 2021.

[19] M. Xu, **Zili Meng**, J. Chen, Y. Guo, C. Sun. Video Playing Method, Video Player and Computer Storage Medium. CN110784760A, Chinese Patent. Granted: August 21, 2020.

1

Introduction

1.1 Research Background and Significance

1.1.1 Real-time Multimedia Transmission

The Internet has become an indispensable part of our lives. Whether it is for work, study, socializing, or entertainment, our daily activities depend on the Internet. In particular, over the past two to three decades, with the continuous upgrading of network technology,

cellular networks have gradually been deployed from 2G to 5G, and wireless local area networks have gradually been deployed from WiFi to WiFi6, greatly enhancing the speed and bandwidth of the Internet. This has led to an increasingly diverse range of Internet applications, extending from traditional text and image transmission to multimedia streaming. Nowadays, it is difficult for people, from urban to rural areas, to imagine life without the Internet. According to statistics, 59.7% of the world's population were long-term Internet users in 2022, with an average monthly data usage of 49.8GB and an average Internet speed of 75.4Mbps[35].

Multimedia streaming applications, which include audio, video, images, text, and various other multimedia data, are an essential component of the Internet. As early as 2016, multimedia traffic accounted for half of the total Internet traffic. By 2022, multimedia traffic had reached 82% of the total Internet traffic [35]. Especially since the outbreak of the COVID-19 pandemic, the real-time nature of multimedia streaming has attracted increasing attention. Tencent Meetings and Zoom software have been extensively applied in various scenarios, such as teaching, conferencing, and remote work. Subsequently, emerging real-time multimedia streaming applications have also garnered widespread attention. Real-time multimedia streaming has expanded into cloud gaming, virtual reality, remote healthcare, and many other areas, extending from traditional person-to-person calls to human-machine interaction control and beyond. Some common scenarios include holographic video conferencing, cloud gaming, virtual reality, remote healthcare, and industrial control, among others.

In summary, Figure 1.1 illustrates the overall structure of real-time multimedia streaming. As multimedia streaming is inherently a network application, we divide it vertically

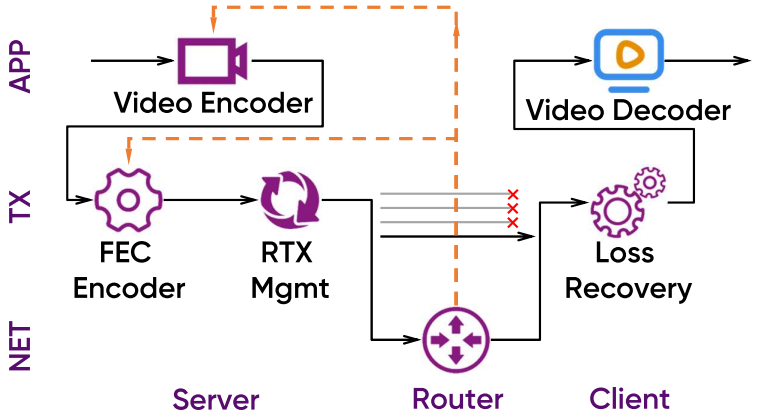


Figure 1.1: Overall structure of real-time multimedia streaming

into application, transport, and network layers from the perspective of the Internet architecture, and horizontally into servers, routers, and clients.

Real-time multimedia transmission has received widespread attention in academia in recent years. High-level international conferences in the fields of networking, such as SIGCOMM and NSDI, and multimedia, such as MM and MMSys, have published numerous papers on optimization in this direction. In the industrial sector, numerous open-source and closed-source frameworks have emerged. Examples include Google's WebRTC, Alibaba's AliRTC, Agora's AgoraRTC, and Tencent's TRTC.

1.1.2 Performance of Real-time Multimedia streaming: Latency Fluctuation

Latency is the most crucial metric for real-time multimedia transmission as it directly correlates with user experience. Real-time multimedia transmission applications not only require low latency but also demand stability in latency. For example, assume that most of the time, wireless users can experience satisfactory round-trip delays of less than 100ms.

However, if the 99th percentile of network round-trip delays exceeds 400ms, the network latency will far surpass the application's latency budget [172, 199]. In this case, one out of every 100 packets may experience high latency, severely affecting the user experience. Therefore, reducing tail latency and stabilizing latency fluctuations are of paramount importance for real-time multimedia transmission applications.

Strict Deadline Requirements

As interactive streaming applications continuously interact with humans, controlling end-to-end latency is essential for achieving a satisfactory user experience. For instance, video conferencing aims for an end-to-end latency of less than 130ms [150, 188], while cloud gaming strives for a latency of less than 96ms [151]*. In practice, server-side and client-side processing typically require approximately 30 ms [45, 123, 239, 259]. Therefore, the end-to-end round-trip delay of the network should not exceed 50-150ms (depending on the application), which constitutes the application's deadline [27, 241].

We conducted a measurement on a typical cloud gaming service (Tencent START Cloud Gaming). During the measurement, the round-trip interaction latency of each video frame was categorized into several intervals. This allowed us to study users' tolerance for different latency: when users experience higher interaction latency and terminate their sessions due to an inability to tolerate such high latency, the frames with high interaction latency will be very close to the end of the user's session. Therefore, this measurement analyzes users' reactions to latency by examining the distribution of frames with different latency. Figure

*This is based on the statistics of most users. Different users and applications may have varying sensitivity to latency. For example, for gaming applications, 3D games have stricter latency requirements than 2D games [143].

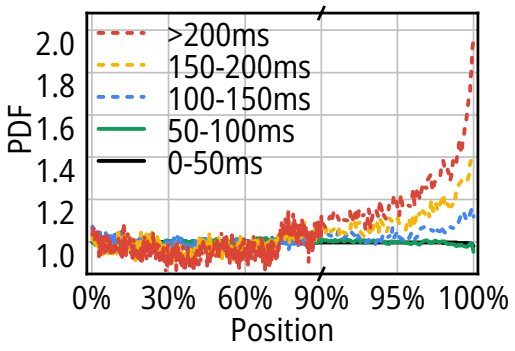


Figure 1.2: Distribution of frame locations for different latencies

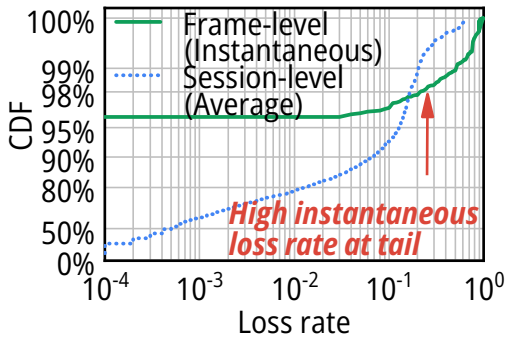


Figure 1.3: Distribution of session-level and frame-level packet loss rates

1.2 shows the distribution of frame positions in the stream for each category, where the x-axis represents the position of the frame in a session, normalized by the total length of the session. For example, a position of 99% indicates that the frame appears very close to the end of the session. If a line is horizontal between 0% and 100% (e.g., the solid lines in the figure), it indicates that these frames appear uniformly throughout the session. Conversely, the three dashed lines in the figure indicate that these frames are more likely to appear at the end of the session. Compared to the uniform distribution of low-latency frames (solid lines), frames with latency greater than 100ms (dashed lines) have a higher probability of appearing at the end of the stream. We infer that this is because users tend to terminate sessions when experiencing higher latency. This also suggests that as long as packets can be transmitted within the deadline (approximately 100ms in this case), faster transmission rates will not significantly impact the user experience.

Therefore, the deadline miss rate (DMR) should be minimized to achieve seamless user experience in real-time multimedia transmission. For example, in the cloud gaming service, the interaction latency deadline is approximately 100ms. For real-time multimedia

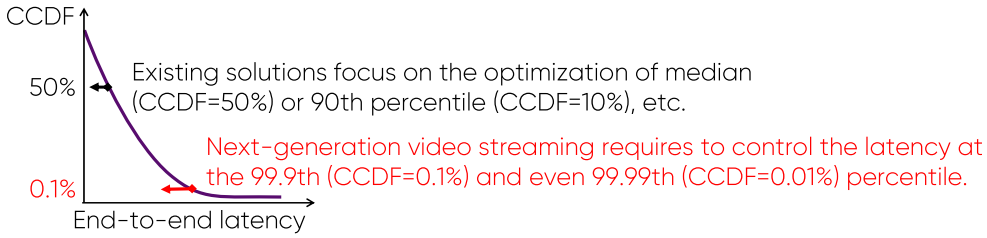


Figure 1.4: This paper focuses more on the optimization of extreme tail latency control for real-time multimedia transmission

Table 1.1: Recent relevant measurement results on wireless network latency

Narayanan et al. (2020) [201]	Tail latency of 5G hops has not improved much compared to 4G, and can be as high as 200ms.
Daldou et al. (2020) [91]	The average WiFi hop latency of 802.11ax (also known as WiFi 6) is greater than 30ms with 30 interferers.
Bhartia et al. (2017) [62]	Up to a quarter of 802.11a wireless access points suffer from >100ms latency on the last hop.
Ghoshal et al. (2022) [124]	For median users, 5G millimeter wave does not improve maximum latency much compared to 4G LTE.

transmission, this deadline miss rate needs to be reduced to an extremely low level. For instance, even if the DMR is 10^{-3}, it would result in a decrease in user experience for one out of every 1000 frames. Note that when the frame rate is 60fps, this interval is a mere 17 seconds. Such an occurrence every tens of seconds would significantly degrade the user experience [27].

This differs from the focus of existing work. Figure 1.4 provides a perspective on latency distribution (complementary cumulative distribution function), with traditional work generally focusing on the more common 50th percentile latency (and sometimes the 90th percentile). However, 10^{-3} implies that we need to focus on the 99.9% percentile latency, which presents a new set of requirements.

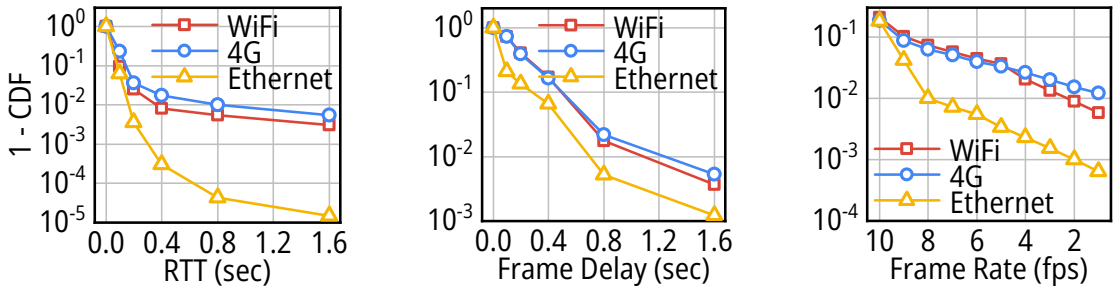


Figure 1.5: Round-trip latency, frame delay, and frame rate distribution on WiFi, 4G, and wired networks

Unsatisfactory Performance

However, the current network performance, especially wireless access network performance, is unsatisfactory at the tail end. Several recent observations support this view. First, existing literature reveals that even when using advanced access technologies, wireless networks exhibit long tail latencies. We summarize recent measurement results in Table 1.1. Even with WiFi 6 (802.11ax) or 5G (millimeter wave), wireless networks still perform poorly. This is consistent with feedback from some content providers. For example, the technical director of a large telecommunications service cloud provider said, "We recommend that customers use wired networks to access cloud desktops." A cloud gaming provider's guide states, "If you encounter network problems, please plug your computer into a wired Ethernet connection if possible" [29]. Latency-sensitive applications find that users prefer inconvenient but stable wired networks due to the high tail latency of wireless networks.

Additionally, our measurements reveal a decline in wireless network tail performance. We measured an online real-time communication service that serves millions of users daily and showed the network conditions and application performance of wired, WiFi, and 4G access networks. The data comes from an online real-time communication service that serves

millions of users daily. Frame delay refers to the latency measured at the application layer. As shown in Figure 1.5, most of the time, wireless networks can provide satisfactory round-trip latencies (less than 100ms). However, the 99th percentile round-trip latency for wireless networks exceeds 400ms, far surpassing the application's latency budget[172, 199]. In this case, one out of every 100 packets may experience high latency, severely affecting the user experience. Application layer metrics show a similar pattern: wireless users encounter twice the video latency (long frame delays) as Ethernet users. Moreover, the frame rate drop (video stutter) ratio for wireless networks is ten times that of wired networks.

1.2 Research Content

This dissertation commences with an examination of internet architecture, conducting a comprehensive analysis of the sources of latency fluctuations in the new generation of multimedia transmission. The study encompasses the complete process of identifying, defining, and resolving the issue. Initially, the research focuses on the latency fluctuations in the new generation of multimedia transmission, conducting a thorough analysis and identifying the impact of multiple links in the internet architecture on latency fluctuations. Subsequently, the study optimizes the causes of latency fluctuations in these links, delving into an in-depth investigation from the application layer to the network layer of the protocol stack.

The following sections provide a brief introduction to the main content of each part of the study.

1.2.1 Review of Related Research

Real-time multimedia transmission applications are vital components of the internet, having been researched for decades. Solutions abound from academia to industry, both domestically and internationally. Since the onset of the pandemic, the usage of real-time multimedia transmission applications has surged, giving rise to numerous cutting-edge research projects. However, there is currently no comprehensive review of the latest research advancements in these newer studies. This dissertation organizes and analyzes these new works according to the various levels of internet architecture while summarizing existing research. The study classifies existing work based on control and data paths, highlighting some shortcomings in current research. The detailed work is presented in Chapter 2.

1.2.2 Real-time Multimedia Transmission Architecture

Before analyzing the latency issues in real-time multimedia transmission, it is essential to understand the sources of latency. This dissertation systematically proposes possible sources of latency fluctuations in real-time multimedia transmission, highlighting the impact of control path and data path latency on overall end-to-end latency. The study first analyzes the lifecycle of a video frame from rendering to playback, breaking down each component and expressing them formally. Additionally, the possible interactions between different components are examined. Through a comprehensive analysis and modeling of the real-time multimedia transmission architecture, the study can optimize each component individually.

In this context, the control path refers to the passage of control information, specifically the control loop's response to performance fluctuations in the network. If the endpoint

responds too late to network fluctuations, latency fluctuations may result from bandwidth mismatches. The data path refers to the route a data packet itself takes, encompassing the application, transport, and network layers. If latency fluctuations occur at any stage, the overall end-to-end latency of the data packet will fluctuate accordingly. The detailed work is presented in Chapter 3.

1.2.3 Control Path Latency

In the analysis and optimization of control path latency, this dissertation focuses on the feedback and decision-making components. The latency fluctuations and reliability of these two components directly affect the end-to-end latency of data packets.

Feedback refers to the process by which the signal of "fluctuation occurrence" in the network reaches the sender. When the network's available bandwidth changes at a specific moment, the sender cannot instantly understand this change – the transmission of this message takes time. For example, in the TCP protocol, messages are usually transmitted through changes in ACK packet delay or delivery rate. In this case, the feedback time requires at least one round-trip delay. The detailed work is presented in Chapter 4.

Decision-making refers to the process from the sender's initial receipt of the network fluctuation signal to the sender's response. If the sender only observes one or two signals of network fluctuations, it may not take them seriously – network noise is substantial, and changes in one or two packets may be mistaken for noise. For example, many congestion control algorithms only make decisions after observing a specific network change for a continuous RTT or even longer. In this case, the timeliness and reliability of decisions are crucial. The detailed work is presented in Chapter 5.

1.2.4 Data Path Latency

In the analysis and optimization of data path latency, this dissertation primarily focuses on the application, transport, and network layers, which are the upper layers in the network architecture. The latency fluctuations in each of these layers directly affect the end-to-end latency of data packets.

Application layer latency generally includes data processing and queuing delays at the application layer. For example, at the receiving end, once the data has been sequenced by the transport layer, if the upper-layer application cannot promptly process and retrieve the data, it must queue at the application layer, waiting for processing. In this situation, if the application's end-to-end latency requirements are stringent (e.g., real-time multimedia transmission), this queuing will result in increased end-to-end latency. This issue becomes increasingly severe as the resolution and frame rate requirements of real-time multimedia transmission increase. Consequently, active management of application layer latency is necessary. The detailed work is presented in Chapter 6.

Transport layer latency is primarily caused by packet loss recovery. A data packet may be lost during transmission, whether due to queue overflow or wireless network interference, resulting in the packet being damaged and unable to pass verification to reach the receiver. In this case, the sender must promptly identify the packet loss event and retransmit the lost data packet. However, both the identification of packet loss (e.g., out-of-order packets and packet loss may appear similar) and the process of retransmitting lost data packets generate additional latency. Additionally, due to the "best-effort" nature of internet design, packet loss is often challenging to avoid. As real-time multimedia transmission latency require-

ments become increasingly stringent, reducing transport layer latency is also necessary. The detailed work is presented in Chapter 7.

Network layer latency is generally caused by mismatches in queue rates and bandwidth. In network queues, multiple users' traffic shares (or competes for) the same resource – bandwidth. The traffic characteristics of other users are often difficult for the current user to predict in advance. For example, if other users competing with a real-time multimedia transmission stream suddenly increase their transmission rate, the available bandwidth for the real-time multimedia transmission traffic will typically decrease. Of course, congestion control algorithms will converge to a new steady state. However, as attention to tail latency increases, even the fluctuations in this convergence process can lead to a decline in end-to-end latency and corresponding user experience. The detailed work is presented in Chapter 8.

1.3 Main Contributions

In conjunction with the overview of the overall research content in the previous section, this dissertation's main contributions are as follows:

1. In the feedback loop of the control path, a solution is proposed to shorten the congestion signal of the feedback loop. This chapter provides a detailed analysis of how end-to-end congestion control and rate control mechanisms in the current network respond to network fluctuations. Through numerous experiments, it is demonstrated that when the feedback loop expands, data path latency fluctuations increase with the expansion of the feedback loop. This is particularly evident in long-distance trans-

mission over wide-area networks (e.g., video conferencing). To address this phenomenon, this chapter proposes an early feedback solution to shorten the congestion signal of the feedback loop by decoupling the feedback loop from the data path. Specifically, this work classifies real-time multimedia transmission protocols based on their feedback modes into in-band and out-of-band feedback and optimizes different feedback modes accordingly. Experiments based on real routers and large-scale simulations show that the proposed early feedback solution for shortening the congestion signal of the feedback loop effectively reduces end-to-end latency fluctuations, thereby enhancing user experience.

2. In the decision-making loop of the control path, a rate control decision framework is proposed, characterized by low decision latency and stable results. This chapter focuses on the analysis of how the complexity of rate control decision algorithms has increased with the introduction of complex rate control decision algorithms, such as deep learning and integer programming, leading to an increase in the computational overhead of end-to-end rate control decision algorithms. Through experiments, it is shown that as decision-making becomes increasingly black-boxed and time-consuming, these complex algorithms may introduce more performance fluctuations in end-to-end rate control decisions due to potential decision lag and errors. To address this phenomenon, this chapter proposes a lightweight, reliable rate control decision transformation and interpretation framework that simplifies complex rate control decision algorithms into simpler ones, achieving timely and reliable decision-making. Specifically, this work converts existing complex rate control decision algorithms based on machine learning and integer programming into simple decision tree-based rate control decision algorithms. Ex-

periments and analysis based on existing algorithms show that the proposed lightweight, reliable rate control decision transformation and interpretation framework effectively reduces performance fluctuations, thereby enhancing user experience.

3. In the application layer of the data path, a solution is proposed to reduce application layer queuing latency through adaptive frame rate adjustment. This chapter primarily analyzes how, with the emergence of next-generation multimedia applications (e.g., cloud gaming), application demands for multimedia transmission quality have increased, leading to a continuous increase in latency for video codecs in the application layer. Through large-scale measurements of real applications, it is shown that when latency fluctuations in the application layer's video codecs are significant, data path latency fluctuations also increase with the latency fluctuations in the video codecs. Since existing application layer designs lack active queue management, this latency is easily further amplified. To address this phenomenon, this chapter proposes an adaptive frame rate adjustment solution that reduces latency fluctuations in video codecs in the application layer by actively adjusting the frame rate. Specifically, this work is based on a joint analysis of network conditions and application conditions, employing queueing theory and stochastic process modeling to develop an active queue management solution for the application layer. Experiments for large-scale users demonstrate that the proposed adaptive frame rate adjustment solution effectively reduces end-to-end latency fluctuations in cloud gaming applications.

4. In the transport layer of the data path, a packet loss recovery mechanism for real-time multimedia transmission is proposed. This chapter emphasizes that as the new generation of multimedia applications becomes less tolerant of latency fluctuations, existing transport layer packet loss recovery mechanisms can no longer meet application demands for latency fluctuations. Analysis based on real measurement data shows that, under existing transport layer packet loss recovery mechanisms and current network conditions, relying solely on retransmission or redundancy for single packet loss recovery is almost unattainable. To address this phenomenon, this chapter proposes a joint packet loss recovery solution that combines existing packet loss recovery mechanisms, particularly retransmission and redundancy recovery. Specifically, this work employs a Markov chain to model packet loss and retransmission jointly, developing an optimal strategy for adding redundancy and determining whether to retransmit. Experiments based on real network datasets show that the proposed joint packet loss recovery solution effectively reduces end-to-end latency fluctuations while also reducing bandwidth costs.

5. In the network layer of the data path, a router queue management scheme is proposed that suppresses multi-application competition queuing and stabilizes performance. This chapter notes that although many solutions attempt to control latency fluctuations at the endpoint, end-to-end latency fluctuations caused by sudden multi-application competition queuing in the network layer remain a serious problem. Regardless of endpoint optimization, the endpoint cannot predict other users' sudden competition in the network, so optimization of bottleneck router management is still necessary to reduce end-to-end latency fluctuations. Measurements based on

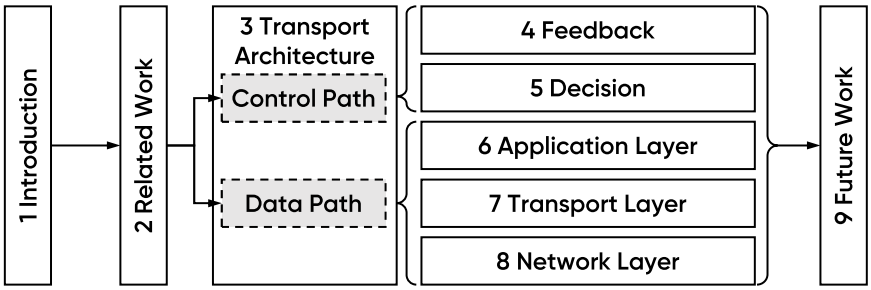


Figure 1.6: Dissertation Roadmap

thousands of websites show that end-to-end latency fluctuations caused by web applications are a significant issue, especially for next-generation multimedia applications with high latency fluctuation requirements. To address this phenomenon, this chapter proposes a novel router queue management scheme that reduces end-to-end latency fluctuations by differentiating service optimization for bandwidth allocation without relying on endpoint information. Specifically, this work observes the encroachment of different flows on bottleneck queues to infer the latency sensitivity of different flows, thereby achieving differentiated service optimization for various flows. Tests based on real routers and thousands of websites show that the proposed router queue management scheme effectively reduces end-to-end latency fluctuations without relying on any endpoint labels or other information.

1.4 Dissertation Roadmap

This dissertation consists of nine chapters, with the overall structure illustrated in Figure 1.6.

Chapter 1 serves as the introduction, presenting the research background, content, main contributions, and thesis organization.

Chapter 2 covers related work, introducing the related research involved in this dissertation.

Chapter 3 outlines the existing real-time multimedia transmission architecture analyzed in this dissertation and examines which components' latency affects the final user-perceived latency. In this chapter, it is noted that the latency of the two components in the control path and the three components in the data path affect the final user-perceived end-to-end latency.

Chapters 4 to 8 address the two paths and five issues mentioned in the previous chapter.

In the control path, Chapters 4 and 5 study the latency fluctuations caused by the two components in the control path: Chapter 4 introduces a solution to reduce latency fluctuations by decreasing the feedback loop in response to the end-to-end latency fluctuations caused by the expansion of the control path feedback latency; Chapter 5 introduces a solution to reduce performance fluctuations in end-to-end rate control decisions by simplifying models in response to the end-to-end performance fluctuations caused by unstable and time-consuming decision-making in the control path.

In the data path, Chapters 6 to 8 study the latency fluctuations caused by the three components in the data path: Chapter 6 introduces a solution to reduce latency fluctuations by adaptively adjusting the frame rate in response to the end-to-end latency fluctuations caused by the performance fluctuations of video codecs in the application layer; Chapter 7 introduces a solution to reduce latency fluctuations by implementing a joint recovery solution that combines multiple packet loss recovery mechanisms in response to the end-to-end

latency fluctuations caused by packet loss and its recovery mechanism in the transport layer; Chapter 8 introduces a solution to reduce latency fluctuations by optimizing differentiated service for bandwidth allocation without relying on endpoint information in response to the end-to-end latency fluctuations caused by sudden multi-application competition queuing in the network layer.

Finally, Chapter 9 concludes the dissertation, summarizing the research results and looking forward to some unfinished research directions.

2

Related Work

This chapter provides an overview of existing research on multimedia transmission systems, with a particular emphasis on network-based optimization for multimedia transmission. The chapter follows the division of existing technologies according to the Internet architecture introduced in the previous section, discussing how the existing work in each layer

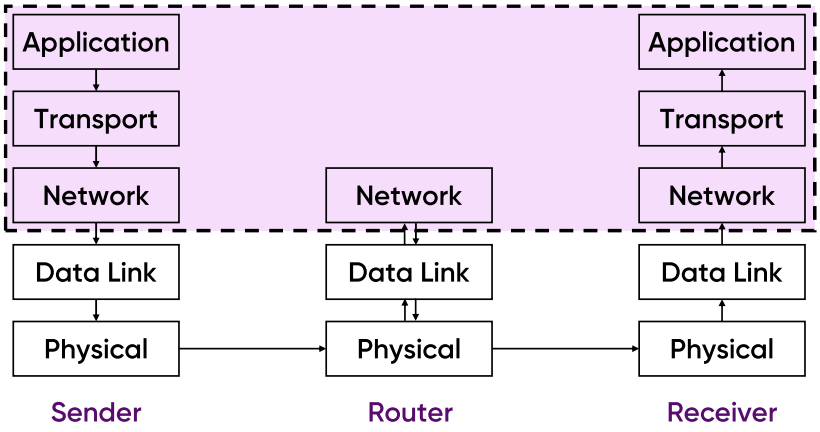


Figure 2.1: Internet architecture and the main focus of related work in this chapter

improves the performance of real-time multimedia transmission or, more generally, reduces network application latency.

As shown in Figure 2.1, this chapter will primarily discuss the application layer, transport layer, and network layer work within the dashed box. These are the areas of primary concern for network researchers. Link layer and physical layer work are more often the focus of researchers in the communication field. In this paper, low-latency work and real-time multimedia work in these two layers will not be discussed. In the following sections, the innovations of this paper will be elaborated upon in comparison to existing work.

Section 2.1 introduces various application layer optimization efforts for multimedia transmission. The first category is optimization for codecs, including the development of new encoding and decoding algorithms to achieve stable performance on fluctuating network links. The second category involves protocol design tailored to multimedia transmission characteristics to better adapt to the multimedia content being transmitted. The

third category is adaptive adjustment of applications based on network conditions, such as adaptive bitrate adjustment, to enhance the user experience.

Section 2.2 presents work on optimizing latency and jitter at the transport layer of the existing Internet. The two main functions of the transport layer are rate control and reliable transmission. The first category is recent work on Internet latency optimization through rate control, represented by congestion control. The second category focuses on latency optimization for multimedia transmission through reliable transmission, represented by packet loss recovery.

Section 2.3 introduces network layer work on controlling latency within the network. The primary devices in the network layer are network routers. The first category of work involves adjusting the buffer size on routers to control latency. The second category discusses how active queue management on routers can be used in conjunction with endpoint algorithms to control latency. The third category directly introduces performance optimization through end-to-end message passing.

2.1 Application Layer on Data Path

Real-time multimedia transmission applications mainly include the following key features: First, when a video source generates an image, the encoder must encode the image into a video stream. For example, video conference images are obtained from the user's camera, while cloud gaming and virtual reality images are rendered by the GPU. All of these require the original video to be encoded before it can be transmitted. Second, during the video encoding process, the encoding parameters (mainly the bitrate) need to be adjusted in real-time based on network conditions. For instance, when the network conditions improve,

Table 2.1: Real-time multimedia optimization related work in the application layer

Academic/Industry Proposals	Solution	Main Approach
Swift (NSDI'22)[92] CGEncoder (MMSys'20)[272] VP9 (Google'13)[74]	Real-time Video Codec	Ensure content decoding in weak network conditions through updated codec design
BB (SIGCOMM'14)[141] Pensieve (SIGCOMM'17)[179] Puffer (NSDI'20)[265]	Adaptive Bitrate Algorithm	Adapt bitrate to bandwidth, reducing buffering
RTP/RTCP (RFC8888) [229] RTSP (RFC7826)[232] DTP (ICNP'21) [277]	Multimedia Transmission Protocol	Protocol design to convey necessary application information

the encoder can increase the encoding bitrate to deliver clearer video content to the user. Similarly, when network conditions worsen, the encoder will reduce the bitrate to ensure that the user can at least see smooth content. Third, once the content is ready, the sender needs to send it using a protocol specific to the application for better content management.

As mentioned in the introduction to this chapter, there has been an increasing amount of work in recent years on multimedia transmission, particularly in the application layer and in conjunction with application design. We will introduce these efforts in the following sections.

2.1.1 Codec Optimization

The history of codec development is long, and its range of applications is extensive. The main principle of video encoding is to exploit the temporal and spatial correlations of video content, significantly compressing the content through differential value storage and other methods to save bandwidth costs. The most widely used codec today is H.264 [4]. In recent years, codec optimization efforts have mainly focused on two aspects: optimizing the

performance of the codec and tailoring the codec design to specific application scenarios. In terms of codec performance optimization, a new H.265 codec mechanism has been introduced in recent years [5], which can save a considerable amount of bandwidth costs at the same level of clarity. However, due to various issues such as patent rights, its deployment is far from ideal compared to H.264. Recently, researchers have also been promoting the standardization and demonstration of H.266 codec applications. In the field of real-time multimedia transmission, the VP9 codec promoted and deployed by Google is currently the most widely used [74]. It is now included in the WebRTC real-time audio and video transmission framework and can be easily used by developers.

In addition, other research efforts have focused on optimizing other application metrics, such as image or video quality (e.g., SSIM [258] or PSNR [14]). For example, Alfalfa [118] specifically optimizes the multi-threaded parallelism of a large number of users during the transcoding process of real-time multimedia transmission. Salsify [119] further makes the encoder aware of network conditions and reserves space for network adjustments. The most recent work, Swift [92], uses neural network techniques to further optimize the encoding efficiency and processing latency of the codec, allowing users to use virtual reality (VR) and other technologies more smoothly. CGEncoder [272] combines the characteristics of cloud gaming and other gaming applications, first analyzing the specific needs of game users – the user experience of game users may not necessarily be entirely consistent with objective indicators of clarity such as PSNR or SSIM. Based on this, it further designs a codec mechanism to make the codec more suitable for cloud gaming scenarios. The work mentioned above is orthogonal to the interaction latency (per-frame latency) that we are concerned with; they focus on video clarity, while we focus on the potential interaction lag

that users may experience. Therefore, the optimizations for clarity mentioned above can coexist with the latency optimizations of this work.

At the same time, the biggest problem in designing codecs is the issue of deployability. Video decoding has a strong demand for real-time performance: when one video frame is played, the next frame should have been decoded and ready to play to avoid user perception of buffering. However, encoding and decoding involve a large number of mathematical operations, which are extremely resource-intensive within the CPU. As a result, in existing solutions, there are usually dedicated encoding and decoding chips within the CPU or graphics card to speed up the decoding process. However, these hardware decoding chips may not necessarily support the emerging encoding and decoding mechanisms mentioned above. In fact, many works themselves mention one of their major shortcomings as not being supported by existing hardware. Therefore, although the H.264 encoding mechanism has been proposed for more than 20 years, it remains the most widely used encoding and decoding mechanism.

2.1.2 Adaptive Bitrate Optimization

As mentioned in the introduction, adaptive bitrate is one of the essential components of existing multimedia transmission mechanisms. Its primary function is to modify the encoding bitrate of the encoder according to the fluctuations in network conditions, ensuring that the encoder's bitrate does not exceed the network's carrying capacity. Since the birth of real-time multimedia transmission, corresponding adaptive bitrate algorithms have been developed. In the past decade, a typical algorithm is the Buffer-based algorithm proposed by Netflix [141]. It innovatively suggests adjusting the multimedia bitrate in the network

by estimating the client-side buffer status in on-demand multimedia videos. When the client buffer is low, it means that the risk of client stuttering increases, and the encoding bitrate needs to be reduced to deliver new content to the client as soon as possible. When the client buffer is high, it means that the encoder can try to explore a higher encoding bitrate to provide better image quality for the user. In this direction, there are also algorithms like BOLA [245], which make decisions based on buffer occupancy but can also perform theoretical analysis based on Lyapunov stability. BOLA is currently the default adaptive bitrate algorithm for the widely used on-demand streaming media framework dash.js. After that, further improvements and enhancements have been made with algorithms like BOLA-E [246], further optimizing buffer-based methods.

In addition, there are sending rate estimation algorithms, such as PANDA [165] and Squad [257], which are similar to congestion control and estimate the most suitable video bitrate for transmission based on network conditions. In parallel, there are many adaptive rate control algorithms based on both buffer and network status. For example, some researchers have proposed using integer programming to systematically model the adaptive bitrate problem, and proposed the RobustMPC algorithm [269] for optimization and solving, obtaining the most suitable bitrate decision for current transmission. In recent years, there has been a trend of using machine learning algorithms to optimize adaptive bitrate algorithms. Pensieve [179], presented at the SIGCOMM 2017 conference, is the first work to use deep neural networks to optimize adaptive bitrate selection. It uses deep reinforcement learning to model the adaptive bitrate problem and designs corresponding state spaces, action spaces, and reward functions, using a series of algorithms to optimize adaptive bitrate algorithms. Subsequently, further optimizations have been made to the structure and

optimization methods of neural networks in works like HotDASH [235] to enhance user experience. This area of work has been a hot topic in academia in recent years. However, there is still a significant gap between current algorithms [179, 269] and some theoretically optimal analyses. Therefore, we believe there is still much room for improvement. As an example, the academic community has been continuously holding competitions to seek better QoE algorithms [267]. In summary, to achieve better performance, continuously optimized methods have been (and will continue to be) proposed [267].

This work does not focus on adaptive bitrate at the application layer, meaning we are orthogonal to it. This is because adaptive bitrate works similarly to congestion control, adjusting the application-side rate to avoid congestion or stuttering in the network or client. In this paper, our work is more concerned with reducing the delays that are inherently in the application layer. Therefore, existing algorithms in the architecture proposed in Chapter 1 do not target delay jitter or extreme tail latency optimization.

2.1.3 Multimedia Transmission Protocol Design

Another important research work in recent years is the design of new multimedia transmission protocols. Application layer protocols are indispensable in the Internet architecture. On top of transport layer protocols, appropriate application layer protocols are needed to ensure the correct transmission of content. The most widely used protocol recently is the RTP/RTCP protocol [229]. RTP and RTCP are a pair of UDP-based protocols, where RTP sends packets from the server to the client to transmit video content, making it a data path protocol; RTCP is responsible for feeding back network status, video status, and other state information from the client to the server, making it a control path protocol. RTCP

constructs Sender Reports (SR) and Receiver Reports (RR) to report sender and receiver information. RTCP also constructs NACK (Negative Acknowledgement) and TWCC (Transport-wide Congestion Control) messages to report packet loss and delay situations to the sender. The sender can selectively decide whether to retransmit certain packets. In this case, such a UDP-based application layer protocol can essentially achieve almost reliable transmission. Both the open-source framework WebRTC [18] and industrial solutions like Zoom [182] or Google Meet [67] use this protocol or its variants for transmission.

In history, there have also been protocols like RTSP [232] for multimedia transmission applications. These protocols are based on reliable transmission protocols like TCP, eliminating the need for ensuring transmission reliability at the application layer. Recently, some researchers have noticed the latency-sensitive requirements of emerging applications and proposed deadline-aware transport protocols (DTP) [238, 277] that carry deadline information for corresponding data packets in the protocol design. In this case, both network devices and receivers can schedule data packets more reasonably based on the deadline information of data packets, maximizing the satisfaction of application requirements.

In contrast, this paper does not propose a new application layer protocol but seeks reliable low-latency on the existing framework. This is because, from a deployability perspective, we want our solution to be as compatible and coexistent with existing frameworks as possible, making it valuable for practical applications. At the same time, existing protocol designs do not explicitly address application latency requirements. The work at the application layer in this paper can coexist with almost all common application layer protocols.

Table 2.2: Related work on real-time multimedia optimization in the transport layer

Academic/Industry Proposals	Solutions	Main Ideas
Sprout (NSDI'13) [261] GCC (MMSys'16) [76] NADA (RFC8698) [289] Scream (RFC8298) [149] Copa (NSDI'18) [47] Vivace (NSDI'18) [100]	Congestion Control	By adapting the sending rate to bandwidth to reduce latency
WebRTC (ICIP'13) [137] AdaptFEC (MM'19) [115] Tambur (NSDI'23) [225] TLP (RFC8985) [86]	Packet Loss Recovery	By reducing retransmissions to reduce end-to-end latency

2.2 Transport Layer on Data Path

The transport layer is a highly focused component in Internet optimization, especially in the networking community represented by SIGCOMM/NSDI. The main function of the transport layer is to ensure that the data delivered by the application layer can reliably reach the receiving end in a timely manner. This is particularly challenging when latency varies, available bandwidth fluctuates, and network packet loss conditions change constantly. Therefore, the two main functions of the transport layer are rate control and reliable transmission. Rate control mainly focuses on congestion control in the actual Internet, while reliable transmission mainly focuses on packet loss recovery. Rate control works more on a macro scale, trying to avoid long queues in the Internet and ensure its efficiency by adjusting congestion windows and other means. Reliable transmission, on the other hand, works more on a micro scale, aiming to deliver one or a few lost packets to the receiving end. Below, we will briefly introduce the related work in these two aspects that are close to the goal of real-time multimedia transmission.

2.2.1 Congestion Control

Congestion control has a history of more than 40 years and is not covered in detail here. Among them, low-latency congestion control has long been a concern for network researchers. Early congestion control algorithms such as Reno[209] and Cubic[129] aimed to improve network resource utilization by occupying queues as much as possible, but this led to an increase in end-to-end latency. In recent years, a more widely used algorithm is BBR, proposed by Google in 2016 [75]. BBR estimates the bottleneck bandwidth and round-trip time of the link to determine how many data packets should be in the network, and sends data packets at this rate. In this way, BBR no longer needs to occupy the bottleneck queue, and can achieve lower latency. In addition, there are algorithms such as Sprout [261], Verus [273], and Copa [47] that further use latency information to perform more accurate rate control for the TCP protocol. For example, Verus[273] is a congestion control algorithm that specifically adapts to the channel fluctuations of cellular networks by adapting latency estimation; Copa[47] adjusts the congestion window on the endpoint by using the signal of latency fluctuations. They can both effectively achieve lower latency.

In the field of real-time audio and video, these algorithms have also been deployed to some extent. For example, Facebook has tested the Copa algorithm in its live streaming business[122] and achieved good results. In addition, there are many congestion control algorithms specifically designed for real-time audio and video applications. For example, Google proposed the GCC [77] algorithm, which is used in the WebRTC framework. The GCC algorithm controls the sending rate by using delay gradient information - measuring the delay of each packet is usually inaccurate, because distinguishing between queue delay and transmission delay has always been a problem for end-to-end congestion con-

trol algorithms. Therefore, the GCC algorithm focuses on the difference in delay between two packets - called the delay gradient - for rate control: when the packet delay is gradually increasing, the bottleneck queue in the network is probably accumulating. At this time, GCC will reduce its sending rate, and vice versa. In addition, Cisco and Ericsson have also proposed NADA [289] and SCREAM [149] algorithms, respectively, to specifically optimize real-time audio and video transmission. They further use some information including Explicit Congestion Notification (ECN) to reduce end-to-end transmission latency.

However, as we introduced in Chapter 1, even though congestion control has made many efforts to control latency jitter, existing algorithms still struggle to achieve satisfactory latency for real-time multimedia transmission applications. Users still suffer from poor network experiences in many situations. On the one hand, this is certainly because applications have increasingly higher demands for latency and smoothness, and on the other hand, it also shows the limitations of purely end-to-end congestion control algorithm optimization. This work aims to explore the new performance bottlenecks in the transport layer and congestion control based on existing work.

2.2.2 Packet Loss Recovery

Packet loss recovery is an important issue in network transmission, and its purpose is to ensure the reliability of transmission when packet loss occurs in the network. A significant feature that distinguishes the TCP protocol from the UDP protocol is its ability to effectively recover lost packets in the kernel. For most applications, including real-time multimedia transmission, the main recovery method when packet loss occurs in the network is retransmitting the lost packet. Retransmitting packets also requires many design consid-

erations, mainly how to determine if a packet is lost rather than out of order or delayed. Initially, TCP used Retransmission Timeout (RTO) to make this judgment - if the original packet's acknowledgment is not received after waiting for a period of time (usually 1 second or 200 milliseconds), the sender will choose to retransmit the packet [230]. Subsequently, the Fast Retransmit mechanism allowed three consecutive identical acknowledgments to quickly trigger retransmission. Recent work has proposed Tail Loss Probe (TLP) [86] and other mechanisms that can still promptly retransmit discarded packets when waiting for three identical acknowledgments takes too long.

Another line of research is to introduce redundancy for packet loss recovery. This approach is also easy to understand: for example, when the sender intends to send three data packets, the sender can encode a fourth packet using Forward Error Correction (FEC) and send all four packets together, fearing that the packets may be lost. In this case, as long as the receiver receives any three packets, it can recover the fourth packet. In this direction, one approach is to use existing FEC technology but dynamically adjust its parameters according to the current network state: when the network packet loss rate is high, the proportion of redundant packets is higher. For example, Bolot[66] and USF[208] algorithms adjust parameters based on the historical packet loss recovery situation and their recovery capabilities. In this direction, there are also strategies for WebRTC's FEC parameters[17] and even recent algorithms that use deep reinforcement learning and other machine learning tools to further predict network status and optimize redundancy parameters[81]. They all achieve good performance in different experimental test environments, effectively recovering lost packets.

In addition to optimizing redundancy parameters, another category of work is to directly design redundancy coding mechanisms. This usually requires strong knowledge of groups, rings, fields, and other mathematical concepts. Many of these works have also been published in information theory-related journals, such as *IEEE Transactions on Information Theory*. More representative works include AdaptFEC [115], coding mechanisms by Fong et al. [116], and coding mechanisms by Krishnan et al. [157]. However, these algorithms are difficult to deploy in practice due to their high complexity. In fact, XOR codes are currently the most widely used redundancy coding mechanisms in real-time multimedia transmission. Even slightly more complex codes like Reed-Solomon (RS) codes are not yet mature.

The main contribution of this work in the transport layer for packet loss recovery scenarios is to jointly optimize the two types of work, redundancy and retransmission. From a practical perspective, this work does not propose a new redundancy coding mechanism but tries to optimize existing coding mechanisms. Another highlight of this work is to optimize the packet loss recovery mechanism from the perspective of latency fluctuations. These contents will be introduced in detail in Chapter 7.

2.3 Network Layer on Data Path

In recent years, there has not been much optimization work on the network layer in wide-area networks. The main reason is that the main component of the network layer is routers within the network. In other scenarios such as data centers, routers (or switches) are replaced more frequently. Therefore, new technologies have the opportunity for faster deployment. However, in wide-area networks, there is almost no situation where a single en-

Table 2.3: Real-time multimedia optimization related work in the network layer

Academic/Industry Proposals	Solutions	Main Ideas
CoDel (CACM'12) [203] RED[168], BLUE[108], GREEN[110], Yellow[173]	Active Queue Management	Drop packets early to force the sender to slow down to avoid over-sending
BDP/n (SIGMETRICS'21) [244] ABS (INFOCOM'22) [251]	Queue Size Optimization	Set appropriate queue size to reduce latency
XCP (SIGCOMM'02) [153] RCP (INFOCOM'08) [249] Kickass (ICNP'16) [112] ABC (NSDI'20) [125]	End-to-End Message Passing	Carry more dimensions of network state for better decision-making

tity controls all devices on a path. Therefore, in the following discussion, the deployability of these works is an important point we focus on.

On routers, what they can do is to operate the packets passing through the router to implicitly or explicitly inform the sender of the current network status. Based on this, the sender can be implicitly informed by active queue management techniques to achieve low latency - when the network status deteriorates, the router can selectively discard some packets; it can also directly adjust the queue size to physically limit its maximum latency - if the buffer is too small, packets have to be discarded, so although the packet loss rate may increase, the latency can also be bounded, which is not necessarily a bad thing for real-time multimedia. Alternatively, new network layer protocols can be explicitly constructed to carry network status information back to the sender for information delivery purposes. This section will review these works from these perspectives.

2.3.1 Active Queue Management

In the network layer, Active Queue Management (AQM) is a common method to control network congestion. There are many AQM algorithms on routers. The earlier active queue management algorithm is RED [113], which informs the current network deterioration by probabilistically random dropping at an early stage. The default active queue management algorithm currently deployed on many edge routers is CoDel, proposed in 2012 [203], which mainly solves the problem that estimating queue length is difficult to adapt to routers with different bandwidths, while using the dwell time in the queue can more accurately control the latency target. In addition, many more AQM algorithms have been proposed, such as SFB [109], Green [110], Yellow [173], Black [79], and AFD [210]. The latest development is the DualQ algorithm, which has just become an IETF RFC in 2023 [231], which is part of the IETF's L4S working group and performs active queue management by classifying data streams into different categories.

In addition, in data centers, there is a large amount of work on managing the queues of data center switches. Examples include PIAS [51], pFabric [42], and SIGCOMM 2022's ABM [40]. However, the biggest difference between these works and active queue management in wide-area networks is that they can assume cooperation between end hosts: an enterprise can control both switches and servers within its data center. This provides great convenience for flow type differentiation, flow size estimation, etc. However, in the Internet, we cannot make such assumptions. If an algorithm prioritizes a certain type of traffic, all Internet users will disguise their traffic as this type of traffic, rendering the mechanism ineffective. In fact, this is one of the reasons why mechanisms such as differentiated services[53] are less widely used in wide-area networks.

In comparison, they also have a common problem of assuming that the end-to-end congestion control algorithm is sensitive to packet loss or ECN marking. However, with the emergence of rate-based or delay-based congestion control algorithms such as Copa and BBR, they are no longer sensitive to packet loss or ECN. Therefore, if it is expected to rely on packet loss to reduce the sending rate of the end-to-end congestion control algorithm, this requires very serious packet loss. For example, BBR does not respond to packet loss rates below 20% in terms of rate. Therefore, there is an urgent need to optimize new active queue management mechanisms for such delay-sensitive congestion control algorithms.

2.3.2 Queue Size Optimization

How to set the bottleneck queue size has always been a difficult problem in network layer management. A small queue can lead to frequent packet loss in the network when dealing with bursty traffic. A large queue, on the other hand, can cause long queuing delays when the rate adjustment is not timely. Therefore, setting the appropriate queue size has always been a concern for network administrators and an important means to reduce end-to-end latency. In this regard, there have been a series of empirical works exploring this issue. For example, in 2019, experts led by Stanford University Professor Nick McKeown organized the Buffer Sizing Workshop to discuss how to set switch queue sizes. In addition, there are many adaptive queue size works, such as ABS [251]. It adjusts the router's queue size adaptively based on the burstiness of network traffic.

Another major work in queue size optimization is theoretical analysis. Initially, scholars proposed that the bottleneck queue size should be no less than the Bandwidth-Delay Product (BDP) to ensure that the congestion control algorithms at the time (e.g., AIMD

or Vegas) could fully utilize the link capacity throughout the entire cycle [3]. Subsequently, in 2004, Appenzeller et al. [44] proposed that, in fact, by utilizing the statistical multiplexing characteristics of different congestion control flows, the bottleneck queue size can be reduced to $BDP/\sqrt{N}$, where N is the number of flows on the switch. In recent years, scholars have pointed out that with the emergence of new congestion control algorithms such as BBR, the bottleneck queue size can be further reduced to BDP/N [244]. As a result, the maximum possible latency on the switch may also continue to decrease.

However, this setting generally only applies to core backbone switches, as they typically have millions of flows. In edge routers (e.g., home wireless routers), there may be only tens or hundreds of flows in most cases. In this case, since N is small, the result is actually trivial. A more serious problem is that in wireless networks, as described in Chapter 1, the bandwidth fluctuations may be quite large, so the queue has to be set very long. This actually leads to the situation where many last-hop routers have very "deep" queue buffers. In this case, the occurrence of high latency is difficult to avoid. The work in this paper is trying to shorten the end-to-end latency without changing this setting.

2.3.3 End-to-End Message Passing

The last category of work is to design new protocols at the network layer to better communicate between end hosts and network devices. Having good message passing can also conveniently control latency because, in an ideal case, if the end host can perfectly replicate the changes in available bandwidth, there will be no congestion due to improper self-adjustment. Typical work in this area is XCP in 2002 [153] and subsequent RCP [249]. They design new protocols that include the bottleneck bandwidth rate in the protocol

header fields to precisely control the sending rate. This also includes some works that may not have designed new protocols but have similarly carried network status information in existing protocols, such as Kickass [112] and ABC [125]. Kickass [112] passes the available bandwidth information of a flow on the current router back to the sender through the size of IP fragments. ABC [125] uses the two bits left over from differentiated services (TOS) that are not widely used on the Internet to mark whether the current router thinks the flow needs to speed up or slow down.

However, the biggest problem with these works is still the lack of deployability. As mentioned at the beginning of this section, there are numerous innovations in the network layer, but very few have been truly deployed in the Internet. The main reason is that it is extremely difficult to modify network devices. The above schemes require modifications to both network devices and end host devices. This is very difficult in practice: end host devices are usually maintained by content providers (such as Baidu, Alibaba, etc.); while network devices are maintained by equipment manufacturers (such as Huawei, H3C, etc.). Coordinating both parties to make changes to achieve performance gains has been proven to be very difficult in the long history of Internet development.

In the design of this paper, we always adhere to the principle of minimizing modifications to devices. The proposed work can be deployed with benefits by modifying only a single network device, without the need for communication and collaboration with other devices. In this way, the work has a certain degree of deployability and has some examples of actual deployment in the current network.

2.4 Summary

This chapter starts from the characteristics of real-time multimedia and low-latency networks, and according to the existing Internet architecture, introduces the efforts made by academia and industry in these two aspects of optimization from the application layer, transport layer, and network layer perspectives. This chapter first introduces the design work on real-time codecs, adaptive bitrate algorithms, and multimedia transport protocols at the application layer, followed by related work on congestion control and packet loss recovery at the transport layer, and finally introduces work on active queue management, router queue size management, and end-to-end collaborative optimization at the network layer. In the process of introduction, this chapter also analyzes the shortcomings of existing work, laying the groundwork for the introduction of the work in this paper.

3

Real-Time Multimedia Streaming Architecture

As described in Chapter 1, when the optimization goal of real-time multimedia transmission applications shifts to tail latency, the main sources of latency may no longer be consistent with the sources of median or average latency in the original architecture. In this

chapter, we focus on analyzing where the end-to-end latency fluctuations come from in the existing real-time multimedia transmission architecture. We will first analyze the sources of latency fluctuations in general, and then analyze the causes of latency fluctuations from the perspectives of control path and data path.

3.1 Analysis of Latency Fluctuation Sources

In traditional real-time multimedia transmission, the main component of latency is network latency - when the physical distance between sender and receiver is still far, and congestion control mechanisms still produce long queues, network latency will occupy most of the latency. However, as described in Chapter 1, with the deployment of edge nodes, improvements in network congestion control mechanisms, network latency is no longer the main component of latency. The current reality is that in many real-time multimedia transmission applications, application service providers can achieve average or median latency as low as 10-15ms through heavy investment. For example, in applications like cloud gaming, service providers deploy servers from a few nodes across the country to several nodes in each province and region. In this case, for most users, there is likely a computing node in their city to provide services. This greatly shortens network latency. Similarly, as access network technology upgrades from 4G to 5G, and WiFi 4 to WiFi 6, the wireless link transmission latency of the last-hop access network has also been greatly improved.

However, when the focus of real-time multimedia transmission shifts to the tail latency of one in a thousand or one in ten thousand, any small fluctuation in any link may cause the end-to-end latency to rise at the 99.99 percentile. The existing real-time multimedia transmission architecture has considered adapting to fluctuations in different network sit-

uations during design, but not enough attention has been paid to the transition and convergence process from one state to another. This is natural - when the latency percentile of concern is at the 50th or even 90th percentile, there is no need to worry about these transient convergence processes. However, when the application focuses on these tail latencies, these transient convergence processes become crucial. Therefore, this section mainly analyzes the possible sources of latency fluctuations when real-time multimedia transmission focuses on the tail latency and the cutoff time miss rate of one in a thousand.

One important source of latency fluctuations found in this work is the presence of control path latency. As mentioned earlier, network conditions are constantly fluctuating, so the response at the endpoint needs to be constantly adjusted based on network conditions. However, due to the presence of the control loop, the response at the endpoint is often delayed. Formally, the response action $a(t)$ at the endpoint at time t is not based on the network state $s(t)$ at time t, but on the network state $s\left(t - \tau_{control}\right)$ at time $t - \tau_{control}$, where $\tau_{control}$ is the control path latency. Therefore, when the network state changes, the response action $a(t)$ at the endpoint often lags behind the change in network state, leading to fluctuations in end-to-end latency.

This becomes very important when the application focus shifts from latency to tail latency. In the past, when the network state changed, the response at the endpoint might be slightly late. But as long as the endpoint can make the correct response, parameters such as sending rate can converge to the new steady-state value. And this transient process is often short-lived, so it does not affect the median or 90th percentile latency. But network fluctuations do occur occasionally. For example, a measurement in Chapter 4 shows that in some

real wireless network data, the probability of network bandwidth dropping to one-fiftieth of the original may be as high as 1%. In this case, the control path latency becomes crucial.

At the same time, the roles of different components in the data path in end-to-end latency also change. With the deployment of edge data centers and the emergence of new access network technologies such as 5G and WiFi 6, network end-to-end latency at the median (in general) can even be achieved at 10-20 milliseconds[197]. In this case, when we observe that the latency of a video frame rises to hundreds of milliseconds, the possible cause is no longer just the long physical distance between the two parties. Latency fluctuations at the application layer, transport layer, and network layer can all lead to instantaneous latency increases.

The remaining two sections of this chapter will analyze these two aspects. Figure 3.1 shows some components in the real-time multimedia transmission architecture that may affect latency jitter after modeling and analysis. In the control path, feedback latency and decision latency will affect the latency of the control path itself. In the data path, latency at the application layer, transport layer, and network layer will also affect the end-to-end data latency in the data path. The several works involved in this paper are also carried out in these two aspects, aiming to systematically solve the problem of latency fluctuations in real-time multimedia transmission.

3.2 Control Path Delay

This section first identifies the significant role of the control path in causing fluctuations in tail-end-to-end delay. The impact of the control path on the delay of real-time multimedia transmission is indirect and only comes into play when the application focuses on tail delay:

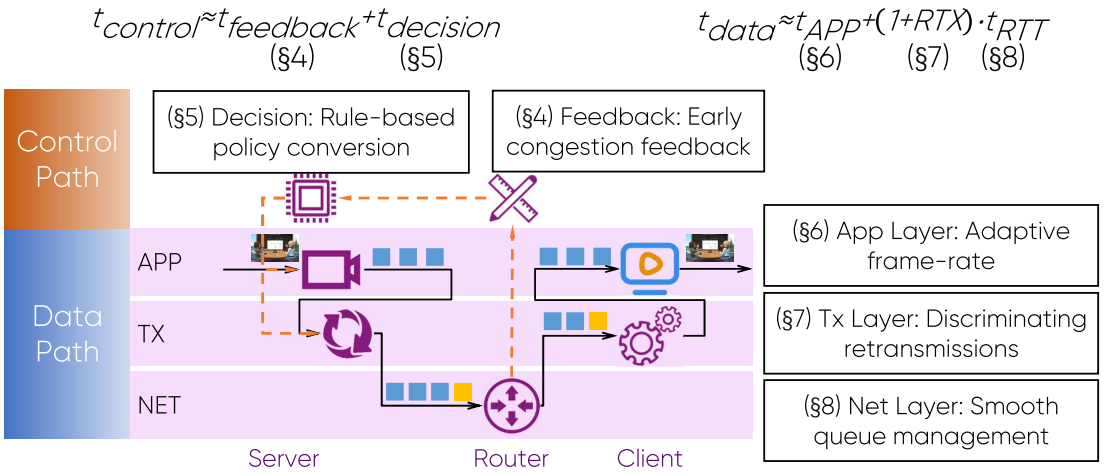


Figure 3.1: Real-time multimedia transmission architecture and the relationship of the works in this paper

if the sending end of the multimedia transmission adjusts its sending rate slower when the network condition changes, this may cause performance degradation due to delay fluctuations. Do not underestimate this little response time: if the sending end needs hundreds of milliseconds to respond each time the network condition changes, the user experience of multimedia transmission in these hundreds of seconds will be poor. However, the Internet is always fluctuating – if the network status fluctuates slightly every few minutes, it means that users have a few thousandths of a chance of encountering performance degradation caused by delay fluctuations. In this case, the delay of the control path becomes crucial.

Figure 3.2 shows an illustrative example. In the Internet, the available bandwidth of a flow may fluctuate at any time due to wireless channel interference and changes in competing traffic patterns. At this time, the sending rate of the sending end of real-time multimedia transmission needs to change accordingly to adapt its throughput to the current available bandwidth in real-time. Without loss of generality, when the available bandwidth

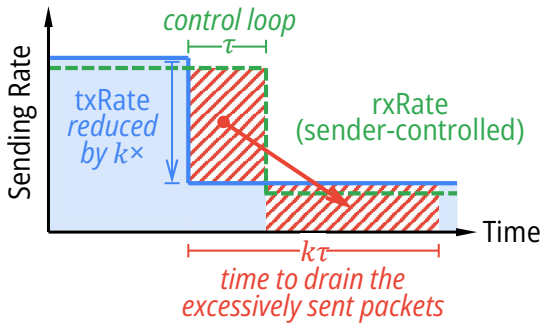


Figure 3.2: An example of control path delay when available bandwidth drops

of a real-time multimedia flow at the bottleneck router (solid line) suddenly drops to one-k of the original, the sending rate of the sending end also needs to be reduced as soon as possible to adapt to the new available bandwidth. However, as mentioned earlier, the decline in available bandwidth cannot be known immediately by the sending end, but requires a control path delay τ (i.e., control loop) to ultimately reflect the decline in sending rate. In this case, the reduction of the sending rate of the sending end will be offset to the right by the reduction of the available bandwidth, as shown by the dashed line in the figure. The solid line in Figure 3.2 is the available bandwidth of the bottleneck router, the dashed line is the sending rate of the bottleneck router, and the red shadow is the backlog of the bottleneck queue. During this time, the bottleneck queue still receives packets at the original sending rate, but its processing rate is greatly reduced due to the drop in available bandwidth. Therefore, these excessive packets will cause a backlog in the bottleneck queue, as shown by the red shadow.

What's worse is that when the control path delay is τ, the time users experience the deterioration of delay is likely to be much more than τ. This is another important observation about the control path delay in this paper — when the network condition fluctuates, due

to the fact that the available bandwidth of the network is actually deteriorating, these packets exceeding the network's carrying capacity need several times the original accumulation time to be cleared. Here, we also analyze this example in Figure 3.2 further. Specifically, the packets arriving at the bottleneck queue need k times the time ($k\tau$) to be sent out during the control loop τ. This is because the data rate sent before the drop in available bandwidth is much higher than the new available bandwidth after the drop. Therefore, the data that may have accumulated in 1 time unit originally takes k time units to alleviate. This is like what is shown in the figure, where the area of the two red shadows in the figure is actually equal. During this time, all sent packets will experience increased delays, thereby reducing user experience.

Specifically, the delay of the control path is divided into two parts: feedback delay and decision delay:

$$t_{control} = t_{feedback} + t_{decision} \tag{3.1}$$

Where the feedback delay $t_{feedback}$ refers to the transmission time of the feedback signal from the sending end to the receiving end, and the decision delay $t_{decision}$ refers to the time for the sending end to make a decision based on the feedback signal. The jitter of these two links may cause fluctuations in the final end-to-end delay. This section first introduces the functions of these two parts and then discusses how they affect the final end-to-end delay fluctuations.

1. End-to-end performance fluctuations caused by feedback delay jitter. How to obtain feedback information is an important problem that almost all control systems face. From circuits, signals to coding adjustments, fault tolerance adjustments, and sending rate

adjustments in real-time multimedia transmission, these are all inseparable from the important role of feedback in decision-making. This is also reflected in the analysis of many existing works in the network field. For example, QCN [41] assumes that the QCN signal sent by the switch in the network needs to be obtained by the sending end after τ time when analyzing stability. Therefore, this feedback delay is actually ubiquitous.

End-to-end control algorithms rely on timely access to network status. For example, TCP congestion control algorithms determine the degree of network congestion based on packet delay, packet loss, and rate changes over a period of time. When network status changes, this change will be immediately reflected in indicators such as packet delay, throughput, and packet loss rate. However, we find that when the network status fluctuates, these indicators often cannot be immediately known by the sending end. In this instant process, the mismatch between the sending rate of the sending end and the network status will cause end-to-end performance fluctuations. In this regard, existing work has the following two shortcomings:

First, existing work generally assumes that the feedback delay is constant. This can greatly simplify many modeling and analysis. However, an important observation in Chapter 4 of this paper is that in the tail case, the feedback delay actually expands with the expansion of the data path delay. This is because, above the network layer, control information does not have a separate control path, but also needs to be transmitted through the data path. Therefore, if the data path causes delay expansion due to packet queuing and other reasons, the feedback delay will also expand with it. This will cause the sending end to know the network status change later, further worsening the end-to-end delay.

Second, existing work mainly focuses on stability rather than performance in analyzing feedback delay. The existence of the feedback loop is essential for many stability analyses: generally speaking, if the feedback loop is too long, so much so that it is longer than the cycle of network status changes, then the control system is likely to be non-convergent. However, in the current Internet, the feedback loop is generally much smaller than the network status change. For example, the general feedback loop is about a round-trip delay, which is about tens of milliseconds in general real-time multimedia transmission applications. Network status generally does not change dramatically every few tens of milliseconds, so the usual analysis results are stable [41]. However, when real-time multimedia transmission focuses on the 99.9th percentile or even later delay requirements, the delay fluctuations in the convergence process will also affect user performance.

Chapter 4 provides a detailed analysis of this and proposes a mechanism to stabilize feedback delay by decoupling the data path and control path. As shown in Figure 3.1, the work in Chapter 4 will mainly optimize the feedback delay $t_{feedback}$ of the control path. The main approach is to decouple control information from the original data packets, unlike traditional protocols that carry control information in the original data packets. In this way, no matter how the data path delay fluctuates and expands, the control path delay can still remain relatively stable. In this case, the sending end can always know the current network status relatively timely and make corresponding adjustments.

2. Rate adjustment lag or error due to unstable decision making. After obtaining network status information, decision-making is another important issue that real-time multimedia transmission control algorithms face. Signals in the network are often full of noise and ambiguity. For example, when the sending end observes a packet loss event, it may

represent that the sending end needs to reduce the sending rate to alleviate network congestion, or it may simply represent a decrease in the channel quality of the wireless link and in fact, the sending end does not need to reduce the sending rate [99]. Therefore, decision-making algorithms tend to make decisions only when enough information is collected to have enough confidence, so that the decisions are reliable enough.

In recent years, due to the continuous pursuit of performance by applications, the decision logic of congestion control, video bitrate adjustment, and other control algorithms has become more and more complex. Starting from traditional heuristic methods that can be implemented in just a few lines of code, researchers have gradually turned to using neural networks [39, 179] or integer programming [269] for decision-making. These attempts are beneficial: on the one hand, researchers' understanding of network links is becoming deeper, and they can make targeted assumptions and modeling for the network. On the other hand, inspired by the progress of neural networks and other emerging machine learning technologies in computer vision and natural language processing, researchers tend to believe that these neural networks have great potential in the field of real-time multimedia transmission. However, the existing decision-making algorithms currently face some problems in the following two dimensions:

First, the decision delay of the decision-making algorithm is increasing. The most direct drawback of using neural networks or integer programming algorithms is their extremely high decision delay. Traditional heuristic algorithms that can be implemented in just a few lines of code hardly consume much decision time when running in actual online deployments. However, even in the forward propagation inference stage, neural networks consume a lot of computing resources. For example, traditional congestion control generally

updates the congestion window every time a packet is received. In a flow with a throughput of 30Mbps (this is a common traffic size for high-definition real-time multimedia such as cloud gaming), this means that the maximum interval between two packets is 0.4 milliseconds. However, the forward prediction of neural networks may still consume milliseconds of delay even if specialized acceleration hardware such as GPUs is used. This phenomenon is more severe in modeling methods such as integer programming optimization. Complex integer programming may take several minutes or even hours to solve. This creates a contradiction between the current high decision delay and the need for high-frequency decision-making in network algorithms.

Second, the decision logic of these algorithms is becoming more and more black-boxed, making it difficult for network administrators to understand the logic behind their decisions. For example, neural networks usually contain thousands (sometimes even billions [70]) of neurons and output their decision results through complex calculations. Administrators generally have difficulty understanding how a decision is made. This leads to a potentially frightening fact – decisions may be wrong and not discovered, so how can network administrators trust such a model? This is similar in optimization algorithms such as integer programming. When the solution result deviates significantly from the administrator's common sense, the administrator has no way of knowing which constraint or variable design is problematic. In this case, the wrong decision will also cause end-to-end performance fluctuations.

Chapter 5 also analyzes the above two problems and proposes an algorithm that can avoid end-to-end performance fluctuations due to unstable decision making by converting complex algorithms into lightweight, stable decision trees. As shown in Figure 3.1, the

work in Chapter 5 will mainly optimize the feedback delay $t_{decision}$ of the control path. The main approach is to decouple offline optimization from online deployment and no longer bind offline optimization with online deployment as in existing work. Network administrators can still use the algorithms and models they think have high performance and good results for optimization when training offline. However, when deploying this optimized model online, the method can convert it into a decision tree model with low decision delay and interpretability with low performance loss. In this case, the delay and reliability of the decision part can be guaranteed in most cases.

3.3 Data Path Delay

In this section, we first qualitatively analyze the components of end-to-end delay in the data path. Under ideal conditions, the delay in the data path is affected by the following factors:

$$t_{data} = t_{app} + (1 + RTX) \times t_{RTT} \tag{3.2}$$

Here, t_{app}, RTX, and t_{RTT} represent the application layer processing time, the number of retransmissions, and the round-trip time, respectively. They correspond to the impact of the application layer, transport layer, and network layer on the end-to-end delay in the data path. Specifically, we first introduce how the (possible) design flaws in these three layers affect the end-to-end delay in the data path.

- t_{app} is the application layer processing time, which is mainly related to the design of the application layer - for example, if the application layer needs to encode video frames, the encoding time will increase.

- RTX is the number of retransmissions, which is mainly related to the design of the transport layer's packet loss recovery - for example, if a packet is lost RTX times, it will arrive at the receiver on the $RTX + 1$ transmission.

- t_{RTT} is the round-trip time, which is mainly related to the design of the network layer's queue management - for example, the longer the queue in the network, the longer the round-trip time.

For example, suppose a data packet is first processed by the encoder at the sender for 5ms (application layer). Then, the data packet starts to be prepared for transmission in the transport layer. The current network RTT is 30ms. Unfortunately, this data packet is always dropped in the network until the 4th transmission when it successfully arrives at the receiver and is acknowledged. Assuming that under ideal conditions, the sender can always determine within 1 RTT that a data packet has been dropped. Therefore, in the transport layer, the total time consumed by this data packet is 30×4=120ms. After arriving at the receiver, there may be some additional delay in the application layer. For example, the decoding of the video may take 10ms. Thus, according to the formula 3.2, the total end-to-end delay is approximately 15ms+120ms=135ms.

Of course, this model has some approximations. For example, in reality, the sender may not be able to detect the loss of a data packet immediately. In fact, TCP requires waiting for the successful arrival and acknowledgment of the next three data packets after the first packet loss to trigger the fast recovery mechanism. After the second loss, it will continue to retransmit after the retransmission timeout (RTO). The above formula is just an estimate of an ideal situation - we can always achieve the ideal situation in the estimate by improving the protocol design, such as the NACK design in the RTP/RTCP of the WebRTC frame-

work. However, the above estimate provides an analysis of the main components of the data path delay.

In recent years, some changes have occurred in these components in the new generation of multimedia transmission. The jitter of these three links will also lead to fluctuations in the final end-to-end delay.

1. Application layer delay: Increased video quality leads to fluctuations in application layer encoding and decoding and network collaboration. In the application layer delay, one significant change we notice is the waiting delay at the interface between the application and the protocol stack (e.g., socket buffer). The existing application layer design does not consider the delay fluctuations that may be introduced by bottlenecks in the application - the current socket buffer in the operating system passively waits for the application to read data from it without actively managing the queue. When the application can handle the data sent by the network in time, the application will actively read data from the buffer. When the application is temporarily unable to handle this data, the data will accumulate in the buffer waiting. As the buffer gradually decreases, the current TCP protocol will adjust the flow control window (advertised window) accordingly to inform the sender to reduce the amount of data sent. When the buffer is full, the receiver will no longer receive new data from the network until the application processes some data to free up space.

However, this design faces an intuitive problem: if the application processing is not timely, the queue will continue to accumulate until it is full. This is similar to the situation in the network where the router's queue, if not actively managed, will accumulate until it overflows, resulting in high queuing delays. This design is not friendly to low-latency applications, including real-time multimedia transmission.

The problem of this queue becomes more severe with the development of multimedia transmission applications. As the new generation of multimedia demands higher image quality, the computational complexity of video encoding and decoding also increases. For example, the resolution has evolved from 240p in the past to 1080p today, and in the future, there may be real-time multimedia transmissions with even higher resolutions such as 4K and 8K. The frame rate of the video has also increased from about 24fps for video calls to 60fps, 90fps, or even higher. This will make the burden of processing data in the application heavier, leading to fluctuations in end-to-end delay.

Based on this, Chapter 6 provides a detailed analysis of the above problem and proposes an application layer queue management mechanism that can actively control the queue in front of the application when a bottleneck occurs, thereby avoiding fluctuations in end-to-end delay. Especially in the increasingly popular high-definition and high-frame-rate real-time multimedia transmission, the adoption of the mechanism proposed in Chapter 6 becomes more and more urgent. As shown in Figure 3.1, the work of Chapter 6 will mainly optimize the application layer delay t_{app} in the data path. The main method is to actively manage this buffer queue instead of waiting for it to overflow passively. Like the active queue management algorithm on the router, when the buffer queue starts to grow, the application layer protocol notifies the sender to reduce the sending rate to avoid the buffer accumulating to a higher position. In this way, the application layer delay can be effectively controlled, making the end-to-end delay fluctuation less.

2. Transport layer delay: The increased demand for delay fluctuation makes the existing transport layer packet loss recovery mechanism unsatisfactory. In the transport layer delay, we notice that when the focus on delay percentile increases from 50th

percentile, 90th percentile to 99.9th percentile, 99.99th percentile, the existing transport layer packet loss recovery mechanism can no longer meet this requirement. Many current transport layer designs do not consider the delay problems that may be caused by small probability tail events. A typical example is packet loss recovery. When data packets are not lost, there are no problems with delay. However, if a data packet is unfortunately lost, the current design of the transport layer is to trigger the fast recovery mechanism after the first time, but it may have to wait for one second to trigger the timeout retransmission after the second time. Although many designs try to speed up this process (e.g., TLP [86]), retransmission is still usually inevitable. One problem with this is that when more extreme situations occur, the delay of data packets in these extreme situations may be very poor. For example, if the instantaneous packet loss rate in the network reaches 10%, a data packet needs to be transmitted 4 times to reach the receiver, then the delay of this data packet will increase by 4 times. And at a packet loss rate of 10%, the probability of a data packet being transmitted 4 times (i.e., being dropped 3 times in a row) is as high as one in a thousand. This will directly affect the user experience at the tail.

It is worth noting that, especially for video frames in real-time multimedia transmission, for a general decoder, a frame can only be delivered to the application for decoding and rendering when all data packets of the frame have arrived at the receiver. That is, if a video frame has 50 data packets, even if one data packet experiences the above situation, the user's experience will be affected by this video frame. Therefore, the existing packet loss recovery mechanism is difficult to meet the extreme requirements of the new generation of multimedia transmission for delay jitter.

This tail problem also becomes more severe with the development of real-time multimedia transmission applications. Game users may be able to tolerate a one-in-a-thousand stutter rate, but applications like remote surgery and remote assisted driving actually require a one-in-ten-thousand or even lower stutter rate. Imagine a complex surgery or a long-distance trip that lasts for hours, and even a few seconds of stuttering can be fatal. These few seconds in ten hours are roughly equivalent to a one-in-ten-thousand or even lower stutter rate requirement. At the same time, as the video transmission bit rate increases, but the data unit (Maximum Transmission Unit, MTU) in the network does not increase accordingly, the phenomenon of a video frame containing multiple data packets and being slowed down by one of the data packets in the aforementioned problem becomes more serious. Therefore, it is important to reconsider these applications that mainly rely on packet loss recovery and did not properly consider tail delay fluctuations during design.

Based on this, Chapter 7 also provides a detailed analysis of the above problem, supported by measurement data, and proposes a transport layer packet loss recovery mechanism. This mechanism can significantly alleviate the end-to-end delay jitter of real-time multimedia transmission caused by packet loss retransmission, even with reduced bandwidth costs. Especially when the pursuit of the tail becomes more and more extreme, the adoption of the mechanism proposed in Chapter 7 becomes more and more urgent. As shown in Figure 3.1, the work of Chapter 7 will mainly optimize the transport layer delay in the data path, that is, its retransmission times RTX. The main method is to combine two common packet loss recovery mechanisms (redundant coding and packet loss retransmission) to increase the intensity of redundant coding when entering the more likely tail situation (e.g., the 2nd and 3rd transmissions) to avoid extremely adverse situations. In this way,

the effective control of end-to-end delay fluctuations caused by small probability events can be achieved.

3. Network layer delay: Diversification of congestion control algorithms leads to performance fluctuations in network layer queue management mechanisms. We notice that in recent years, the traffic characteristics and application requirements faced by network layer queue management mechanisms have also changed. As mentioned earlier, the main source of delay in the network layer is queuing. The cause of queuing is generally due to the mismatch between the arrival rate and the sending rate of the queue. In traditional network layer queue management, algorithms like CoDel [203] limit the length of the queue to avoid excessive delay due to long queues and have been widely deployed. However, these algorithms face the following two problems under the traffic characteristics of today's network transmission:

First, the response of queue management algorithms is more focused on packet loss rather than other indicators. Current queue management algorithms, when designed initially, targeted more traditional TCP congestion control algorithms, such as Reno [209] and CUBIC [129]. The most prominent feature of these algorithms is that they rely heavily on packet loss signals (or ECN signals) to adjust the rate. For example, when the sender does not observe packet loss, it will continue to try to increase the sending rate (or congestion window). When the sender observes packet loss, it will reduce the sending rate. However, low-latency applications, represented by real-time multimedia transmission, no longer use packet loss-sensitive congestion control algorithms like Cubic, but use delay-sensitive congestion control algorithms like GCC to achieve low latency. This means that existing queue management algorithms may not necessarily be effective in controlling de-

lay: delay-sensitive congestion control algorithms no longer respond to packet loss signals. This makes ensuring low latency not so simple.

Second, the design of queue management algorithms focuses more on macro performance rather than micro performance. Current queue management algorithms focus more on macro performance on a long time scale when measuring performance. For example, in terms of fairness, researchers measure throughput fairness indicators (e.g., Jain's fairness index) on a relatively long time scale. However, they do not pay much attention to how to gradually converge to this fairness in the transient state. In this case, when the focus of performance shifts to tail delay, as mentioned earlier, the transient performance of this convergence process is equally critical. Especially when some competing traffic in the network (e.g., web browsing) has also undergone some new changes, the network layer queue management mechanism can hardly guarantee that the delay jitter of multimedia transmission is within an acceptable range. If, like existing methods, the performance fluctuations in the transient state are not considered, users will suffer a poor experience at the tail.

Based on this, Chapter 8 analyzes the transient performance of queue management algorithms and the response signals of congestion control algorithms and proposes an active queue management mechanism within the network layer that controls performance fluctuations by limiting the instantaneous interference of burst traffic on stable traffic. We find that as web design becomes more complex and application performance requirements become more diverse, the mechanism in Chapter 8 will play an increasingly important role. As shown in Figure 3.1, the work of Chapter 8 will mainly optimize the network layer delay t_{net} in the data path. The main approach is, on the one hand, to no longer rely on packet loss signals to convey information to congestion control algorithms, but to use delay; on

the other hand, to adopt a smooth transition strategy in transient state transitions, allowing congestion control algorithms to have sufficient time to respond to changes in network conditions. In this way, the queue management mechanism can effectively control delay fluctuations when network traffic and other fluctuations occur.

3.4 Summary

This chapter provides a detailed analysis of the architecture of real-time multimedia transmission applications and the components of end-to-end delay. According to the division of control path and data path in this paper, this chapter introduces the fluctuations of feedback and decision-making in the control path and the components of end-to-end delay in the data path, including application layer, transport layer, and network layer. In each specific component, this chapter briefly introduces the main existing problems and the solutions to be proposed in this paper.

4

Feedback on Control Path: Early Congestion Feedback

4.1 Introduction

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

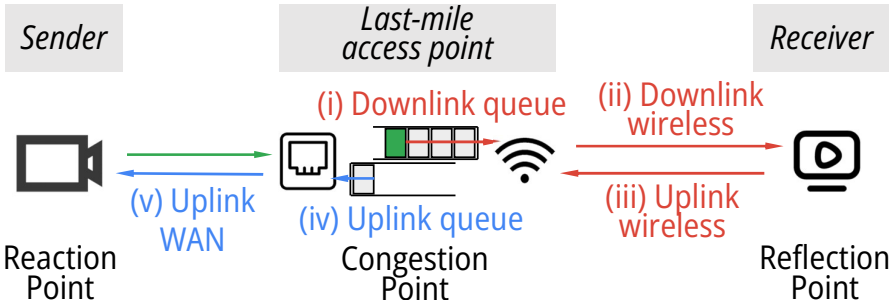


Figure 4.1: Control loop for rate adaption at the wireless last mile. Compared with existing solutions, Zhuge bypasses the segment (i) - (iii) to achieve the shortest control loop.

wireless networks can drop by 10× at the 99th percentile (§4.2.3). 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 4.1, 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 chapter 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 [261] queues. A carefully designed AP, on observing

a filling downlink queue (i in Figure 4.1) can modify or delay packets in the uplink queue (iv in Figure 4.1), 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 [153] or active network [106]). However, coordinating AP and senders that are usually maintained by different entities (§4.2.3) 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 [75]; some protocols are concerned with minimum RTT values over a particular window [47]; and some protocols react to inter-packet timings and are not concerned with RTT at all [77]. 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* that achieves consistent low latency† 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 (§4.4.1) 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 §4.5.2) derived from the Fortune Teller.

We have implemented Zhuge in both simulation and with a WiFi-router based testbed (§4.7). 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)

*Zhuge is a famous fortune-teller in ancient China.

†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 [129]) are not our target.

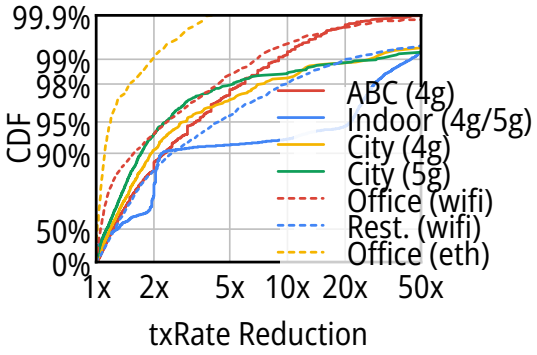


Figure 4.2: Distribution of wireless available bandwidth reduction ratio.

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.

4.2 Background and Motivation

In this section, we use real-world statistics to reveal the status of wireless tail latency (§4.2.1). Next, we analyze why existing solutions fail to achieve consistent low latency (§4.2.2). Finally, we present our motivation of reducing the control loop to ameliorate tail latency (§4.2.3).

4.2.1 Understanding Wireless Tail Latency

We first answer the following one questions: 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. As analyzed in §3.2, 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. Solid lines represent traces from several open datasets, and dashed lines represent traces from our own measurements in the office and restaurant (details in §4.7.2). The available bandwidth is the average value of each 200 ms measurement window. Considering the typical RTT of Internet (tens of milliseconds), the congestion control algorithms should react to fluctuations in such a time scale.

As shown in Figure 4.2, 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.

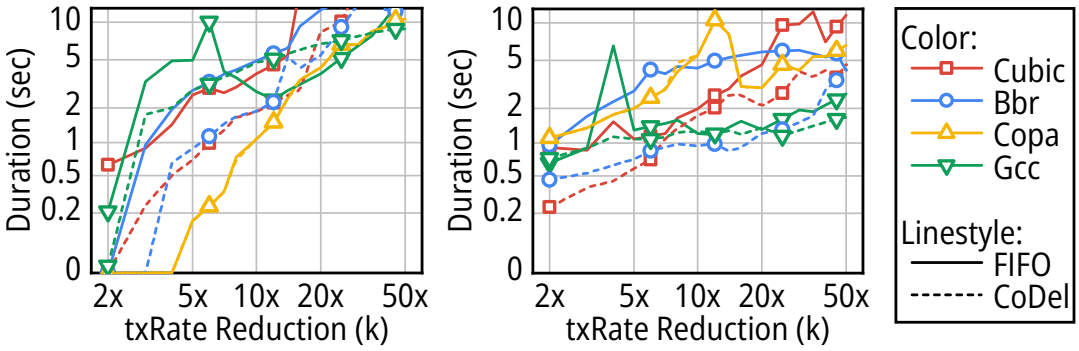
4.2.2 Existing Solutions

The reduction of ABW (k) is due to contention in the link layer and below [152] 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 [75] moves the working point of congestion control from a full queue in CUBIC [129] to an empty queue. CoDel [203] queue management also tries to shorten the queue in the steady state in a variety of network conditions compared with FIFO. Sub-

sequent research efforts (including congestion control [47, 77, 100] and active queue management [136]) 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 4.1, 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 [75], Copa [47], and GCC [77]) together with AQMs in Figure 4.3. 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 [203] drops the packets in the front of the queue to reduce the downlink queuing latency (i) in Figure 4.1, yet still suffers long wireless latency (ii) and (iii), which could be more than 100ms [62]. Moreover, AQMs are mostly designed to *drop* some packets, while many modern CCAs are designed to be re-



(a) RTT degradation.

(b) CWND (sending rate) reduction.

Figure 4.3: 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.*

sponsive to the increase of packet *delay* and insensitive to packet drops [47, 75, 77]. This can also be validated in Figure 4.3(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 [153], RCP [249], Kickass [112], and ABC [125]. 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 §4.7.

4.2.3 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 4.1). 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 1.5.

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 [153], RCP [249], Kickass [112], ABC [125], and active network [106] 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.

4.3 Zhuge Design

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

4.3.1 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 con-

ditions 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 [89]. 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* [171]: The dequeuing rate is usually measured over a sliding window (e.g., 40ms for WiFi in [125]). 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 [16]. For WebRTC-based applications, network conditions are periodically summarized into a special feedback packet [229]. Various CCAs make it challenging to effectively deliver the network conditions to the sender.

4.3.2 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 §4.4 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 estima-

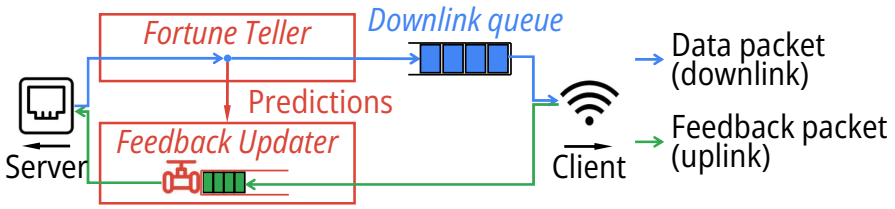


Figure 4.4: The overall workflow of Zhuge at the last-mile AP. Zhuge contributes the Fortune Teller and Feedback Updater.

tors. We measure the average dequeuing rate 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 §4.5 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 [138]. 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 4.4. 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, bypass-

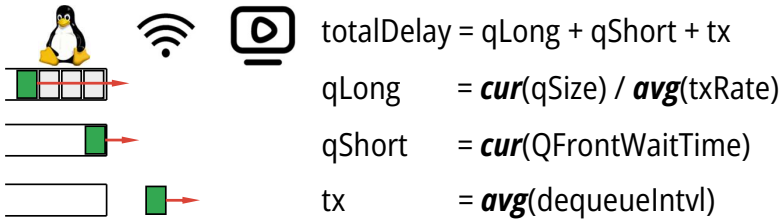


Figure 4.5: Different delay components that the Fortune Teller will estimate. `qLong` and `qShort` together form the queuing delay at the network layer. `tx` is the transmission delay at the link layer.

ing the queuing delay and wireless transmission delay of the control loop (part (i)-(iii) in Figure 4.1).

4.4 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 [135], 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.

4.4.1 Queuing Delay Prediction

As discussed in §4.3.1, 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 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 4.5. 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

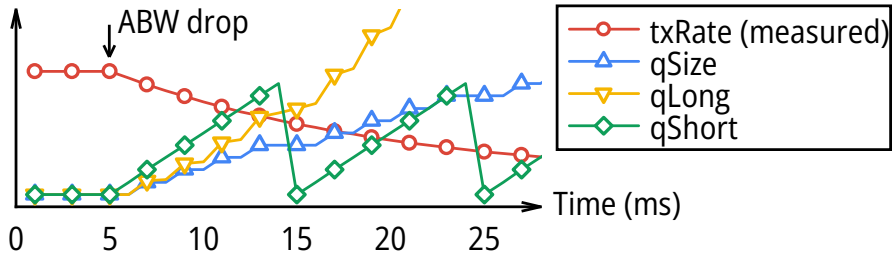


Figure 4.6: How qLong and qShort react to the ABW drop at 5ms.

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 4.5, $avg(\cdot)$ 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 4.6. 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 4.6, 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 4.6), 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

$$\text{qSize} = max(\text{sizeOfPacketsInQueue} - \text{maxBurstSize}, 0) \tag{4.1}$$

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 [125]. For example, the default queue discipline in `systemd` has been changed to `fq_codel` among different flows differentiated by their 5-tuples [8]. For cellular networks, each flow also has its own queue isolated from competing flows [125]. In these cases, we need to calculate the statistics of the RTC flow's corresponding queue.

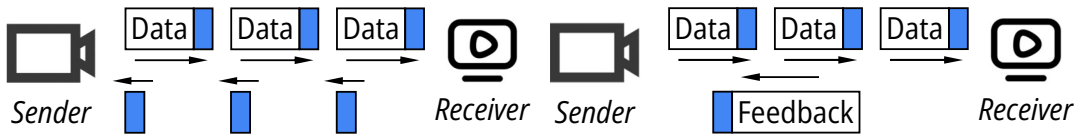
4.4.2 Transmission Delay Prediction

In this paper, we mainly target at the estimation of delays in the WiFi network. We refer the readers to [125] 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 [62]. For example, many Netgear routers adopt the Qualcomm Atheros hardware [6], 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 [125], 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 [135]. 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 4.5, 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.



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

Figure 4.7: 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 (§4.5.2)	TCP QUIC [144]	PCC [100] BBR [75] Copa [47]	Meta Live [122] Windows 365 [155] Twitch [237] Tencent Start [16]
In-band (§4.5.3)	RTP+RTCP [229]	GCC [77] NADA [289] Scream [149]	Google Stadia [96] Zoom [182] Microsoft Teams [224]

Table 4.1: 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.

4.5 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 (§4.5.1), and then introduce our corresponding solutions (§4.5.2 and §4.5.3).

4.5.1 Feedback Mechanism Classification

We investigate popular RTC applications and summarize their feedback mechanisms in Table 4.1. They can be categorized into two types, in-band and out-of-band. We present their behaviors in Figure 4.7.[‡]

- *In-band feedback.* As shown in Figure 4.7(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 [138].

- *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

[‡]Some protocols may utilize both feedback mechanisms. For example, the RTP sender also measures the RTT itself, similar to TCP [229]. This RTT information is not used for rate control, but is only used to stabilize the control loop in RTP.

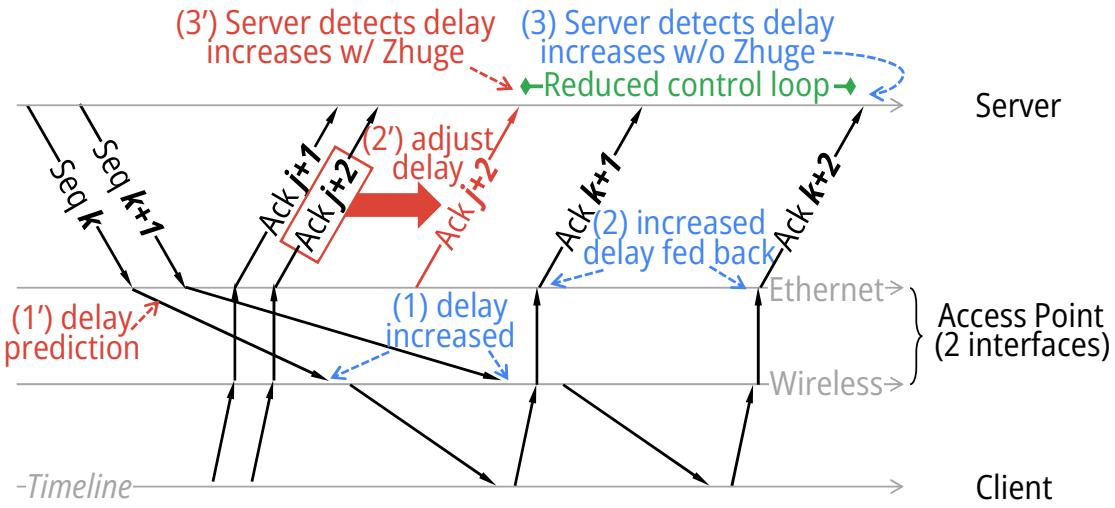


Figure 4.8: Zhuge immediately delays the feedback packets in the reverse direction to carry the predicted fortunes back.

written in the payload of feedback packets, we need to update the payload of feedback packets. Next we introduce two solutions in detail.

4.5.2 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 [47] is sensitive to 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 4.8. 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 4.9, 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 4.8, 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 4.9. Consequently, the control loop of CCAs is reduced by $(k+1)-(j+1)$ (counted in ACK number, the green arrow in Figure 4.8). 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).

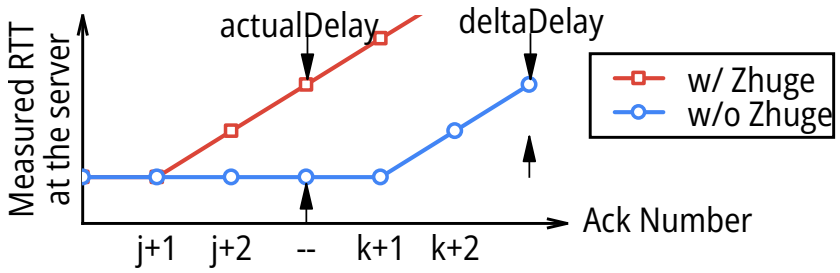


Figure 4.9: Zhuge shifts the curve of RTT forward by delaying earlier returning ACK packet to quickly feedback network conditions. The *actualDelay* is the control loop of Zhuge.

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 §4.4. The final predicted total delay is calculated as:

$$\text{totalDelay} = \text{qLong} + \text{qShort} + \text{tx} \tag{4.2}$$

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

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 4.8, 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

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

```
deltaDelay = curTotalDelay - lastTotalDelay
if deltaDelay ⩾ 0 then
    deltaHistory.push_back(deltaDelay)
else
    tokenHistory.push_back(-deltaDelay)
lastTotalDelay = curtotalDelay
```

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 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

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
```

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).

4.5.3 In-band Feedback: Updating Payloads

For in-band feedback mechanisms such as RTCP [229], 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 [138]), 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 [138, 229]. 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 [60]. 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 [60]. Zhuge might intercept and save the public key of the server, and use it to encrypt the constructed feedback.

4.6 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 [183]. We further evaluate the fairness of Zhuge in §4.7.6.

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.

4.7 Evaluation

We first introduce our implementation of Zhuge in §4.7.1 and the experimental setup in §4.7.2. 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%. (§4.7.3)
- *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. (§4.7.4)
- *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%. (§4.7.5)
- *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. (§4.7.6)

4.7.1 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 §4.7.2. 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 [26, 33]. We use a Netgear WNDR 3800 router [6] 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.

4.7.2 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) [77] 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 [203] 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 [47] is a latency-sensitive CCA for TCP. It can achieve low latency according to many experiments [39, 125] and is already deployed in real streaming services [122].

- *Copa+FastAck.* FastAck [62] 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 [125] 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 [28] 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.1.
- *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 [187].
- *C2 - City 4G* and *C3 - City 5G.* Literature [264] 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.

- *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 [14], SSIM [258], and VMAF [166] since they do not reflect the end-to-end interactive delay. Some recent efforts are focused on subjective experience metrics [81], which is left for our future work.

4.7.3 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 4.10, 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 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

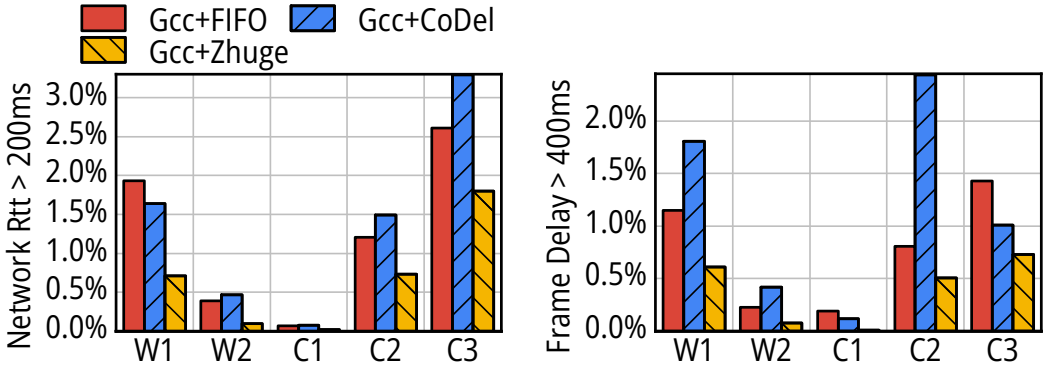
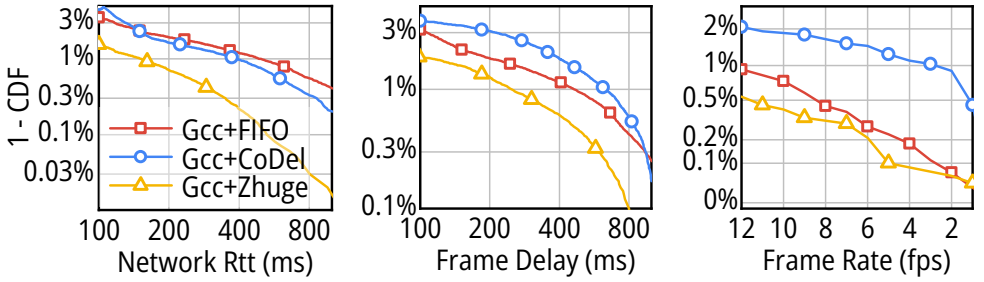
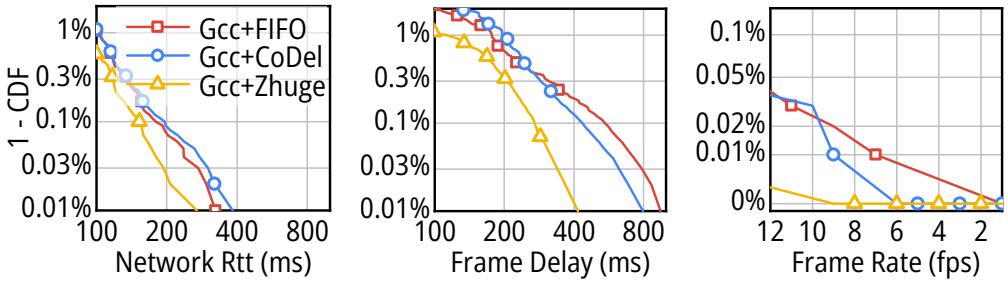


Figure 4.10: Results of trace-driven simulations over RTP/RTCP.



(a) Trace W1 - Restaurant WiFi



(b) Trace C1 - Indoor Mixed 4G/5G

Figure 4.11: Delay distributions of Zhuge and different baselines over RTP/RTCP. Note that all y-axes are log-scaled.

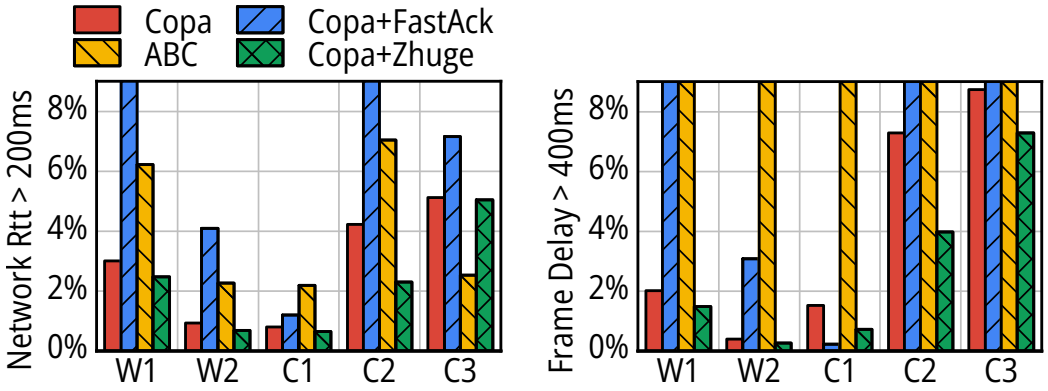
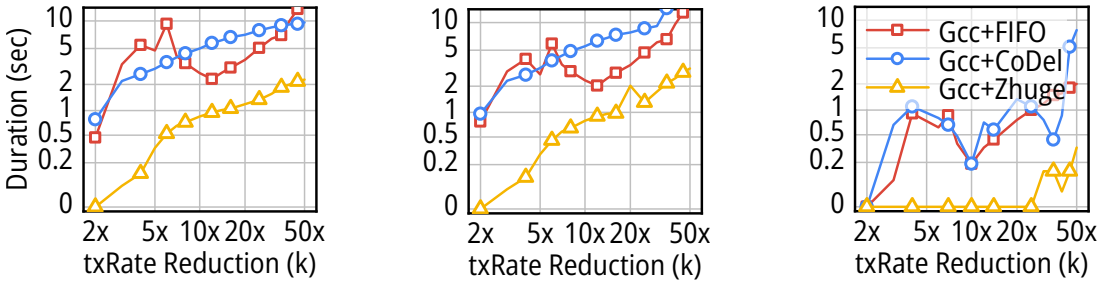


Figure 4.12: Results of trace-driven simulations over TCP.

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 $>10\%$).

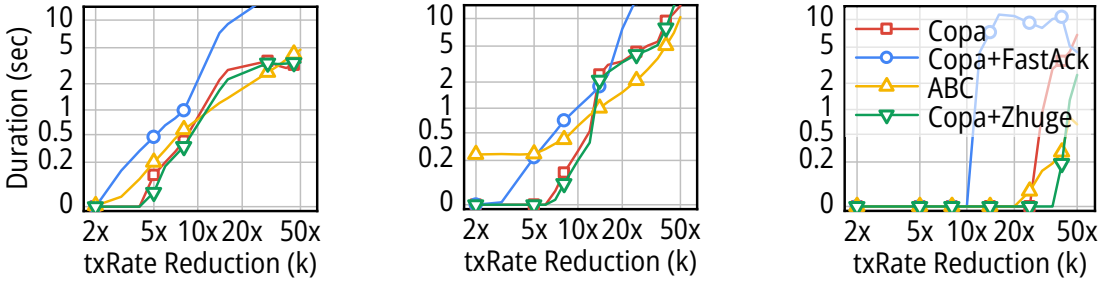
We further present the detailed results of RTP/RTCP based on trace W1 (WiFi) and C1 (cellular) in Figure 4.11 to better understand the optimization of Zhuge. We observe that Zhuge could reduce the tail latency, long frame delay ratio, and low frame rate ratio at all tail percentiles against two baselines. 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 4.12 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 de-



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

Figure 4.13: Performance comparison over RTP under ABW drop.



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

Figure 4.14: Performance comparison over TCP under ABW drop.

ployable at scale as discussed in §4.2.3. 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 A.2 and find that Zhuge also achieves comparable performance with ABC.

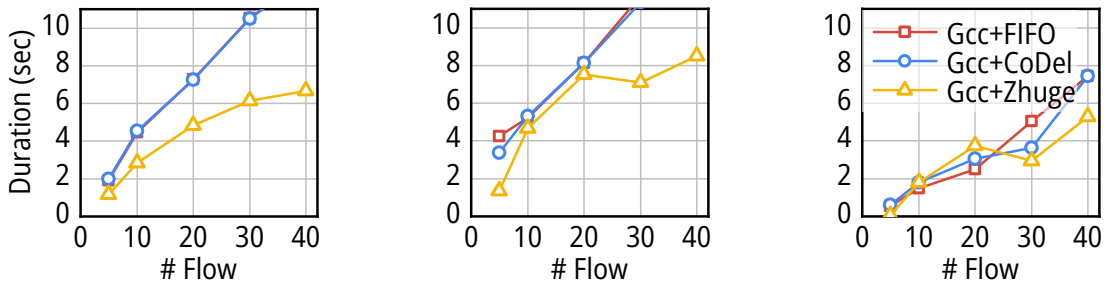
4.7.4 Microbenchmarks under Wireless Fluctuations

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. 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.

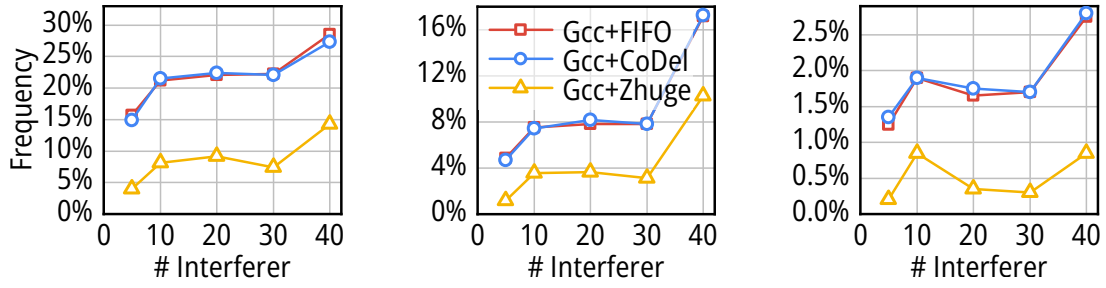
As shown in Figure 4.13, for RTP/RTCP, Gcc+Zhuge reduces the duration of network degradations and application performance by at least 50% in a wide range of settings. Results over TCP show similar results as presented in Figure 4.14. Compared with the better performer of Copa and Copa+FastAck, Copa+Zhuge 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 4.2, 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 4.15 shows that compared with FIFO and CoDel, Zhuge could reduce the



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

Figure 4.15: Performance comparison over RTP under competition.

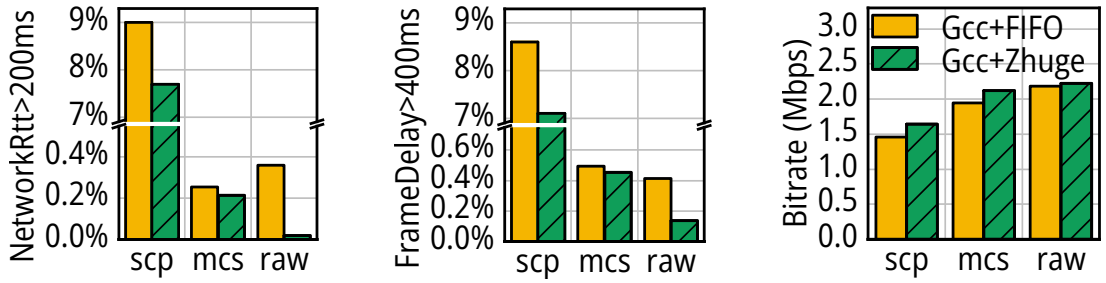


(a) NetworkRtt>200ms (b) FrameDelay>400ms (c) FrameRate<10fps

Figure 4.16: Performance comparison over RTP under interference.

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



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

Figure 4.17: Testbed experiments of Zhuge with an RTC flow.

for a single event as in previous two scenarios. As shown in Figure 4.16, 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 [62], 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.

4.7.5 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 [31] 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 [125], we randomly change the MCS every 30 seconds with the Linux `iw` command and assess Zhuge's reaction to fluctuation.

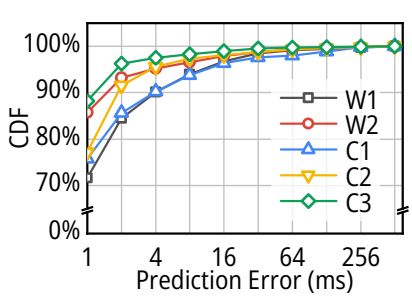
- `raw`. We report the results of running the RTC application in our crowded office without additional configurations.

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 4.17(a) and 4.17(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.

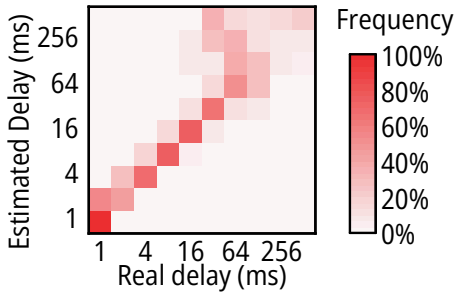
Meanwhile, we also evaluate the capability of Zhuge to maintain similar performance in a steady wireless channel compared with the baseline. We evaluate the steady-state performance by measuring the video's average bitrate based on Microsoft Edge and present the results in Figure 4.17(c). We observe 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.

4.7.6 Zhuge Deep Dive

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



(a) Prediction error by trace.



(b) Heatmap (normalized in each row).

Figure 4.18: Prediction accuracy of Zhuge Fortune Teller.

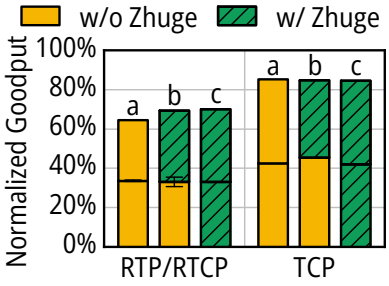


Figure 4.19: Fairness of Zhuge.

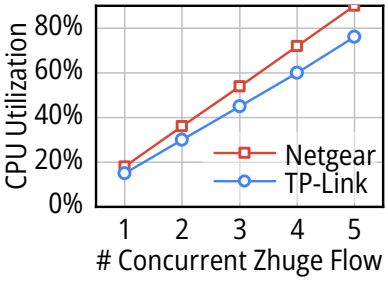


Figure 4.20: CPU Overhead.

Estimation accuracy. We measure the accuracy in the estimation of packet delay in §4.4.1. 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 4.18(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 4.18(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 4.19 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 4.19. For both GCC and Copa, the bitrate difference of the two flows are < 3%. Thus, as discussed in §4.6, 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 [7] AP. We measure the CPU utilization when processing different numbers of concurrent unencrypted RTC flows by Zhuge, and present the result in Figure 4.20. 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 [62, 156].

4.8 Summary

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.

5

Decision on Control Path: Rule-based Policy Conversion

5.1 Introduction

Recent years have witnessed a steady trend of applying deep learning (DL) to a diverse set of network optimization problems, including video streaming [179, 180, 266], local traf-

fic control [83, 147], and network resource management [226, 263, 278]. The key enabler for this trend is the use of Deep Neural Networks (DNNs), thanks to their strong ability to fit complex functions for prediction [160, 162]. Moreover, DNNs are easy to marry with standard optimization techniques such as reinforcement learning (RL) [248] to allow data-driven and automatic performance improvement. Consequently, prior work has demonstrated significant improvement with DNNs over hand-crafted heuristics in multiple network applications [83, 179, 181].

However, the superior performance of DNNs comes at the cost of using millions or even billions of parameters [70, 160]. This cost is fundamentally rooted in the design of DNNs, as they typically require numerous parameters to achieve universal function approximation [162]. Therefore, network operators have to consider DNNs as large black-boxes [94, 283], which makes DL-based networking systems incomprehensible to debug, heavyweight to deploy, and extremely difficult to ad-hoc adjust (§5.2.1). As a result, network operators firmly hold a general fear against using DL-based networking systems for critical deployment in practice.

Over the years, the machine learning community has developed several techniques for understanding the behaviors of DNNs in the scope of image recognition [59, 275] and language translation [218, 252]. These techniques focus on surgically monitoring the activation of neurons to determine the set of features that the neurons are sensitive to [59]. However, directly applying these techniques to DL-based networking systems is not suitable — network operators typically seek simple, deterministic control rules mapped from the input (e.g., scheduling packets with certain headers to a port), as opposed to nitpicking the operational details of DNNs. Besides, networking systems are diverse in terms of

their application settings and their input data structure. The current DNN interpretation tools, designed primarily for well-structured vector inputs (e.g., images, sentences), are not sufficient across diverse networking systems. Therefore, an interpretable DL framework specifically tailored for the networking domain is much needed.

In this chapter, our high-level design goal is to interpret DL-based networking systems with human-readable control policies so that network operators can easily debug, deploy, and ad-hoc adjust DL-based networking systems. We develop Metis*, a general framework that contains two techniques to provide interpretability. To support a wide range of networking systems, Metis finds that a common feature shared by video streaming systems is that they are *local systems*, which collect information locally and make decisions for one instance only.

Specifically, we adopt a decision tree conversion method [58, 223] for local systems. The main observation behind the design choice is that existing heuristic video streaming systems are usually *rule-based* decision-making systems (§5.3.1) with a rather simple decision logic (e.g., buffer-based bitrate adaption (ABR) [141].) The conversion is built atop a teacher-student training process, where the DNN policy acts as the teacher and generates input-output samples to construct the student decision tree [223]. However, to match the performance with DNNs, traditional decision tree algorithms [121] usually output an exceedingly large number of branches, which are effectively uninterpretable. We leverage two important observations to prune the branches down to a tractable number for network operators. First, sensible policies in local systems often unanimously output the same control action for a large part of the observed states. For example, any performant ABR policies [179]

*Metis is a Greek deity that offers wisdom and consultation.

would keep a low bitrate when both of the bandwidth and the playback buffer are low. By relying on the data generated by the teacher DNN, the decision tree can easily cut down the decision space. Second, different input-output pairs have different contributions to the performance of a policy. We adopt a special resampling method [58] that allows the teacher DNN to guide the decision tree to prioritize the actions leading to the best outcome. Empirically, our decision tree can generate human-readable interpretations (§5.5.1), and the performance degradation is within 2% of the original DNNs (§5.5.5).

For concrete evaluation, we generate interpretable policies for DL-based adaptive video streaming systems with Metis (§5.5.1). For example, we interpret the bitrate adaptation policy of Pensieve [179] and recommend a new decision variable. We also present three use cases of Metis in the design, debugging, and deployment of DL-based networking systems. (i) Metis helps network operators to redesign the DNN structure of Pensieve with a quality of experience (QoE) improvement by 5.1%[†] on average (§5.5.3). (ii) Metis debugs the DNN in Pensieve and improves the average QoE by up to 4% with only decision trees (§5.5.4). (iii) Metis enables a lightweight DL-based flow scheduler (AuTO [83]) and a lightweight Pensieve with shorter decision latency by 27× and lower resource consumption by up to 156× (§5.5.5).

We make the following contributions in this paper:

- Metis, a framework to provide interpretation for two general categories of DL-based networking systems, where it interprets video streaming systems with decision trees (§5.3).

[†]Even a 1% improvement in QoE is significant to current Internet video providers (e.g., YouTube) considering the volume of videos [180].

- Prototype implementations of Metis over DL-based video streaming systems Pensieve [179] (§5.4), and their interpretations with capturing well-known heuristics and discovering new knowledge (§5.5.1).
- Three use cases on how Metis can help network operators to design (§5.5.3), and debug (§5.5.4), deploy (§5.5.5), DL-based video streaming systems.

To the best of our knowledge, Metis is the first general framework to interpret diverse DL-based networking systems at deployment. The source code of Metis is available at https://github.com/transys-project/metis/. We believe that Metis will accelerate the deployment of DL-based networking systems in practice.

5.2 Motivation

We motivate the design of Metis by analyzing (i) the drawbacks of current DL-based networking systems (§5.2.1), and (ii) why existing interpretation methods are insufficient for DL-based networking systems (§5.2.2).

5.2.1 Drawbacks of Current Systems

The blackbox property of DNNs lacks interpretability for network operators. Without understanding why DNNs make decisions, network operators might not have enough confidence to adopt them in practice [283]. Moreover, as shown in Figure 5.1, the blackbox property brings drawbacks to networking systems in debugging, online deployment, and ad-hoc adjustment due to the following reasons.

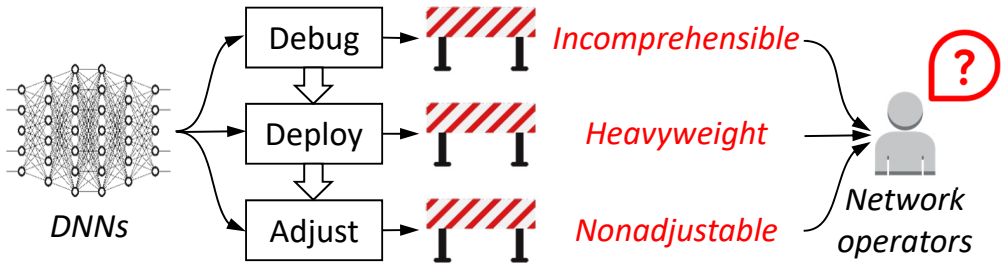


Figure 5.1: DNNs create barriers for network operators in many stages of the development flow of networking systems.

Incomprehensible structure. DNNs could contain thousands to billions of neurons [70], making them incomprehensible for human network operators. Due to the complex structure of DNN, when DL-based networking systems fail to perform as expected, network operators will have difficulty in locating the erroneous component. Even after finding the sub-optimality in the design of DNN structures, network operators are challenged to redesign them for better performance. If network operators could trace the mapping function between inputs and outputs, it would be easier to debug and improve DL-based networking systems.

Heavyweight to deploy. DNNs are known to be bulky on both resource consumption and decision latency [142]. Even with advanced hardware (e.g., GPU), DNNs may take tens of milliseconds for decision-making (§5.5.5). In contrast, networking systems, especially local systems on end devices (e.g., mobile phones) or in-network devices (e.g., switches), are resource-limited and latency-sensitive [142]. For example, loading a DNN-based ABR algorithm on mobile clients increases the page load time by around 10 seconds (§5.5.5), which will make users leave the page. Existing systems usually provide "best-

effort" services only and roll back to heuristics when resource and latency constraints can not be met [83], which degrades the performance of DNNs.

Nonadjustable policies. Practical deployment of networking systems also requires ad-hoc adjustments or adding temporary features. For example, we could adjust the weights for different jobs in fair scheduling to catch up with the fluctuations in workloads [181]. However, the lack of interpretation brings difficulties to network operators when they need to adjust the networking systems. Without understanding why DNNs make such decisions, arbitrary adjustments may lead to severe performance degradation. For example, when network operators want to manually reroute a flow away from a link, without interpretations of decisions, network operators might not know how and where to accommodate that flow.

Discussions. The application of DNNs in networking systems is still at a preliminary stage: DNNs in Pensieve [179], AuTO [83], and RouteNet [226] (published in 2017, 2018, and 2019) have less than ten layers. As a comparison, a sharp increase in the number of DNN layers has been observed in other communities (Figure 5.2). Recent language translation models even contain billions of parameters [70]. Although we are not saying that the larger is the better, it is indisputable that larger DNNs will aggravate the problems and create barriers to deploy DL-based networking systems in practice.

5.2.2 Why Not Existing Interpretations?

For DL-based networking systems, existing interpretation methods [101, 127] are insufficient in the following aspects:

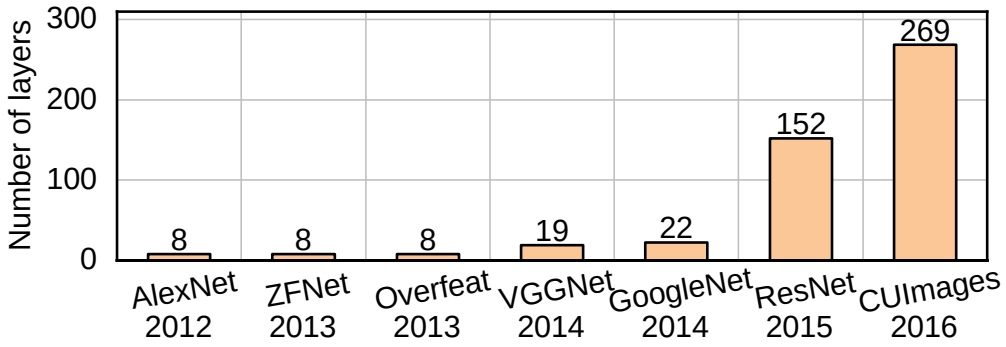


Figure 5.2: The exponential growth of DNN complexity in ImageNet Challenge winners [93] (Figure adopted from [103]).

Different interpretation goal. The question of *why a DNN makes a certain decision* may have answers from two angles. In the machine learning community, the answer could be understanding the *inner mechanism of ML models* (e.g., which neurons are activated for some particular input features) [59, 275]. It's like trying to understand how the brain works with surgery. In contrast, the expected answer from network operators is the *relationship between inputs and outputs* (e.g., which input features affect the decision) [283]. What network operators need is a method to interpret the mapping between the input and output for DNNs.

Diverse networking systems. DL-based networking systems have different application scenarios and are based on various DL approaches, such as feedforward neural network (FNN) [179], recurrent neural network (RNN) [268], and graph neural network (GNN) [181]. Therefore, interpreting diverse DL-based networking systems with one single interpretation method is insufficient. For example, LEMNA [128] could only interpret the behaviors of RNN and thus is not suitable for GNN-based networking systems [181]. In Metis, we

observe that DL-based adaptive video streaming systems are actually local systems and develop corresponding techniques.

In response, to interpret DL-based networking systems, Metis introduces a decision tree-based method for DL-based adaptive video streaming systems.

5.3 Decision Tree Interpretations

In this section, we first describe the design choice for choosing decision trees in Metis (§5.3.1), and then explain the detailed methodology to convert the DNNs to decision trees (§5.3.2).

5.3.1 Design Choice: Decision Tree

As introduced in §5.1, Metis converts DNNs into simpler models based on interpretation methods. There are many candidate models, such as (super)linear regression [128, 217], decision trees [58, 223], etc. We refer the readers to [101, 127] for a comprehensive review.

In this chapter, we decide to convert DNNs to *decision trees* due to three reasons. First, the logic structure of decision trees resembles the policies made by networking systems, which are rule-based policies. For example, ABR algorithms depend on precomputed rules over buffer occupancy and predicted throughput [245, 269]. Second, as shown in Figure 5.3, decision trees have rich expressiveness and high faithfulness because they are non-parametric and can represent very complex policies [65]. We demonstrate the performance of decision trees during conversion compared to other methods [128, 217] in Appendix B.4. Third, decision trees are lightweight for networking systems, which will bring further benefits to resource consumption and decision latency (§5.5.5). There are also re-

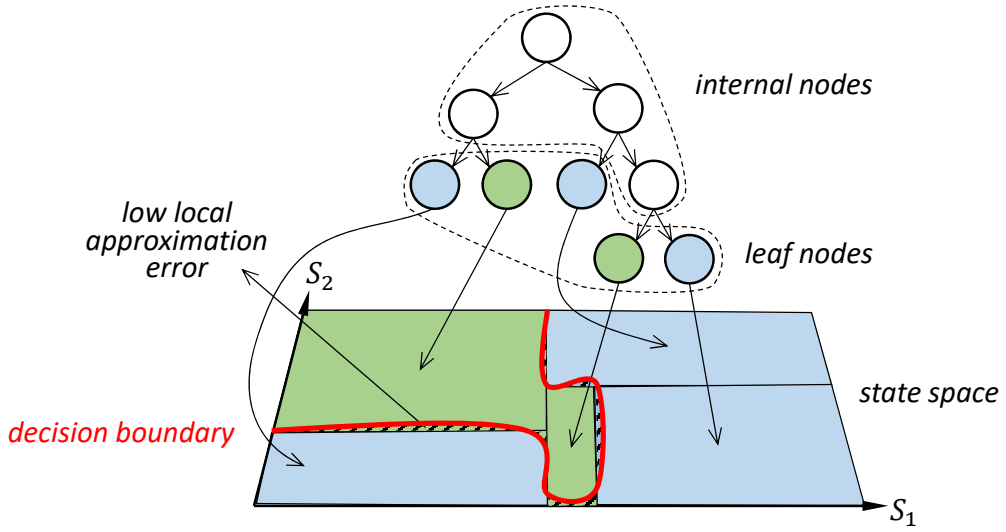


Figure 5.3: An illustration of decision tree approximating the original decision boundary.

search efforts that interpret DNNs with programming language [255, 288]. However, designing different primitives for each networking system is time-consuming and inefficient.

With interpretations in the form of decision trees, we can interpret the results since the decision-making process is transparent (§5.5.1). Also, we can debug the DNN models when they generate sub-optimal decisions (§5.5.4). Furthermore, since decision trees are much smaller in size, less expensive on computation, we could also deploy the decision trees online instead of deploying heavyweight DNN models. This will result in low decision-making latency and resource consumption (§5.5.5).

5.3.2 Conversion Methodology

To extract the decision tree from a finetuned DNN, we adopt a teacher-student training methodology proposed in [58]. Without teacher-student learning, one wrong prediction may drive the student off teacher's trajectory in the state space. As shown in Figure 5.4, a

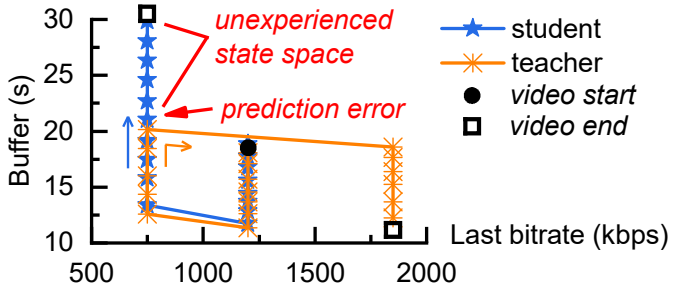


Figure 5.4: An illustration of how teachers correct students.

wrong decision may bring the decision tree into a region of unexperienced state space. The decision tree might thus make more mistakes since it has no prior knowledge about that region of state space. Those mistakes will further drive the decision tree off the trajectory and worsen the performance. In response, Metis continuously simulates the decision tree and lets the original ABR algorithm (teacher) correct the decisions made by that decision tree (student). The decision tree will gradually learn how to make decisions in the whole state space.

We reproduce key conversion steps for networking systems as follows:

Step 1: Traces collection. When training decision trees, it is important to obtain an appropriate dataset from DNNs. Simply covering all possible (state, action) pairs is too costly and does not faithfully reflect the state distribution from the target policy. Thus, Metis follows the trajectories generated by the teacher DNNs. Moreover, networking systems are sequential decision processes, where each action has long-lasting effects on future states. Therefore, the decision tree can deviate significantly from the trajectories of DNNs due to imperfect conversion [58]. To make the converted policy more robust, we let the DNN policy take over the control on the deviated trajectory and re-collect (state, action) pair to

refine the conversion training. We iterate the process until the deviation is confined (i.e., the converted policy closely tracks the DNN trajectory).

Step 2: Resampling. Local systems usually optimize *policies* instead of independent actions [83, 147, 179]. In this case, different actions of networking systems may have different importance to the optimization goal. For example, an ABR algorithm downloading a huge chunk at extremely low buffer will lead to a long stall, resulting in severe performance degradation. Meanwhile, downloading a little larger chunk when network condition and buffer are moderate will not have drastic effects. However, decision tree algorithms are designed to optimize the accuracy of a single action and treat all actions the same. Therefore, their optimization goals do not match. Existing DL-based local systems adopt reinforcement learning (RL) to optimize the policy instead of single actions, where the *advantage* of each (state, action) represents the importance to the optimization goal. Therefore, we follow recent advances in converting DNNs in RL policies into decision trees [58] and resample $\mathcal{D}$ according to the advantage function. For each pair (s, a), the sampling probability $p(s, a)$ could be expressed as:

$$p(s,a) \propto \left(V^{(\pi^*)}(s) - \min_{a' \in A} Q^{(\pi^*)}(s,a') \right) \cdot 1\left((s,a) \in \mathcal{D}\right) \tag{5.1}$$

where $V(s)$ and $Q(s, a)$ are the value function and Q-function of RL [248]. Value function represents the expected total reward starting at state s and following the policy π. Q-function further specifies the next step action a. π^* is the DNN policy, and A is the action space. $1(x)$ is the indicator function, which equals to 1 if and only if x is true. We analyze Equation 5.1 with more details in Appendix B.1. We then retrain the decision tree on the

resampled dataset. Our empirical results demonstrate that the resampling step can improve the QoE over 73% of the traces (Appendix B.1).

Step 3: Pruning. As the size of the decision tree sometimes becomes much larger than network operators can understand, we adopt cost complexity pruning (CCP) [121] to reduce the number of branches according to the requirements from network operators. Compared with other pruning methods, CCP empirically achieves a smaller decision tree with a similar error rate [192]. At its core, CCP creates a cost function of the complexity of the pruned decision tree to balance between accuracy and complexity. Moreover, for the continuous outputs in networking systems (e.g., queue thresholds [83]), we employ the design of the *regression tree* to generate real value outputs [254]. In our experiments, for Pensieve, the size of leaf nodes may be up to 1000 without pruning (Appendix B.5). With CCP, pruning the decision tree down to 200 leaf nodes only results in a performance degradation of less than 0.6% (§5.5.5).

Step 4: Deployment. Finally, network operators could deploy the converted model online and enjoy both the performance improvement brought by deep learning and the interpretability provided by the converted model. Our evaluation shows that the performance degradation of decision trees is less than 2% for two DL-based networking systems (§5.5.5). We also present further benefits of converting DNNs of networking systems into decision trees (easy debugging and lightweight deployment) in §5.5.4 and §5.5.5.

5.4 Implementation

In current Internet video transmissions, each video consists of many *chunks* (a few seconds of playtime), and each chunk is encoded at multiple bitrates. Pensieve [179] is a deep RL-based ABR system to optimize bitrates with network observations such as past chunk throughput, buffer occupancy.

We use the same video in Pensieve unless other specified. The chunk size, bitrates of the video are respectively set to 4 seconds and {300, 750, 1200, 1850, 2850, 4300} kbps. Real-world network traces include 250 HSDPA traces [219] and 205 FCC traces [9]. We integrate DNNs into JavaScript with `tf.js` [242] to run Pensieve in the browser. We set up the same environment and QoE metric with Pensieve.

We then implement Metis +Pensieve. We use the finetuned model provided by [179] to generate the decision tree. We use five baseline ABRs (BB [141], RB [179], Festive [148], BOLA [245], rMPC [269]) as Pensieve and migrate them into `dash.js` [11].

5.5 Experiments

In this section, we first empirically evaluate the interpretability of Metis with two types of DL-based networking systems. Subsequently, we showcase how Metis addresses the drawbacks of existing DL-based networking systems (§5.2.1). We finally benchmark the interpretability of Metis. Overall, our experiments cover the following aspects:

- **System interpretations.** We demonstrate the effectiveness of Metis by presenting the interpretations of Pensieve with newly discovered knowledge (§5.5.1).

- **Performance Maintenance.** We demonstrate the capability of Metis in maintaining the performance as the original model (§5.5.2).
- **Guide for model design.** We present a case on how to improve the DNN structure of Pensieve for better performance based on the interpretations of Metis (§5.5.3).
- **Enabling debuggability.** With a use case of Pensieve, Metis debugs a problem and improves its performance by adjusting the structure of decision trees (§5.5.4).
- **Lightweight deployment.** For Pensieve, network operators could directly deploy the converted decision trees provided by Metis online and achieve benefits enabled by lightweight deployments (§5.5.5).
- **Metis deep dive.** We finally evaluate the interpretation performance, parameter sensitivity, and computation overhead of Metis under different settings (§5.5.6).

5.5.1 System Interpretations

With Metis, we interpret the DNN policy learned by Pensieve. We present the top 4 layers of the decision tree of Metis +Pensieve in Figure 5.5. The decision variables of each node include the last chunk bitrate (r^t), previous throughput (θ^t), buffer occupancy (B), and last chunk download time (T_t). Since we only present the top 4 layers of the decision tree, we represent the frequency of final decisions of each node with the color on the palette in Figure 5.5.

From the interpretations in Figure 5.5, we can know the reasons behind the superior performance of Pensieve in two directions. (i) *Discovering new knowledge.* On the top two

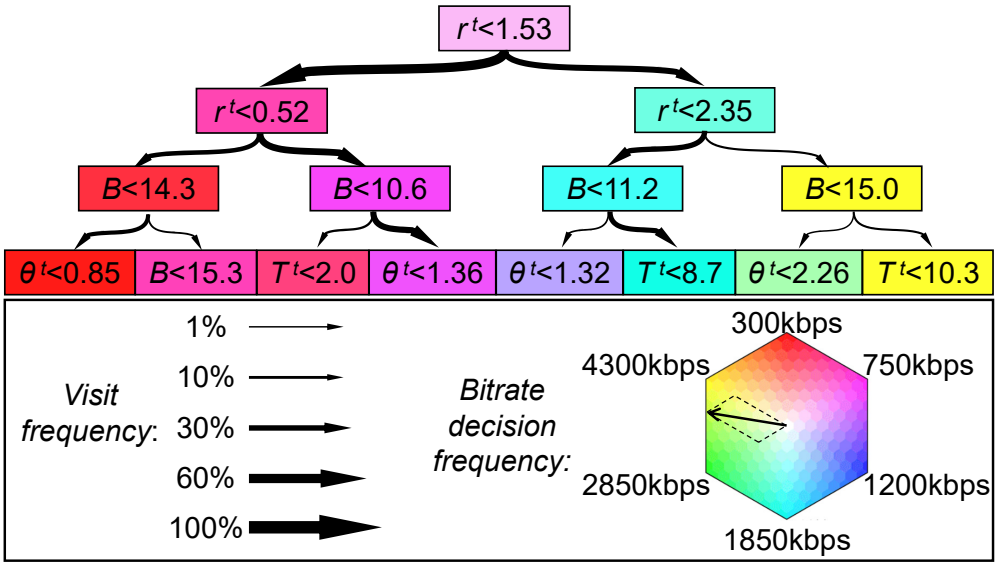


Figure 5.5: Top 4 layers of the decision tree of Metis +Pensieve. The color represents the frequency of bitrate selections at that node. For example, the arrow in the palette represents that 67% states traversing a node with that color are finally decided as 4300kbps, and 33% states are 2850kbps. Better viewed with color.

layers, Metis +Pensieve first classifies inputs into four branches based on the *last chunk bitrate*, which is different from existing methods. The information contained in the last bitrate choice affects the output QoE significantly. Based on this observation, we recommend that network operators could improve ABR algorithms with particular focus on the last chunk bitrate. We present a use case on how to utilize this observation to improve the DNN structure in §5.5.3. (ii) *Capturing existing heuristics.* Similar to existing methods, Metis +Pensieve makes decisions based on buffer occupancy [141, 245] and predicted throughput [11, 269]. With the interpretations provided by Metis, network operators can understand how Pensieve makes decisions.

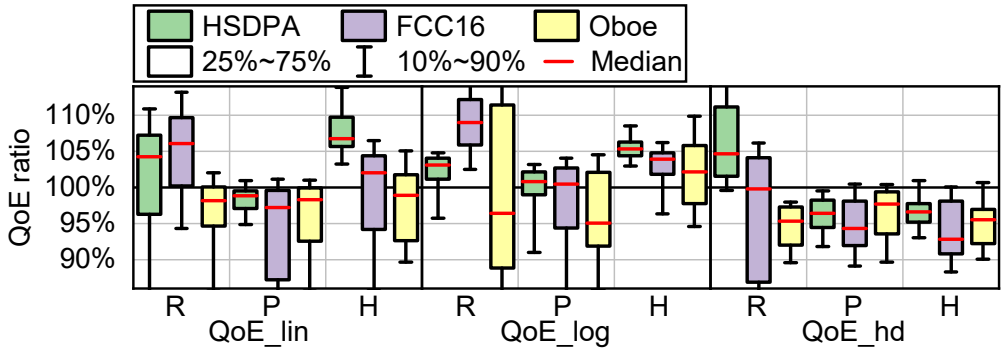


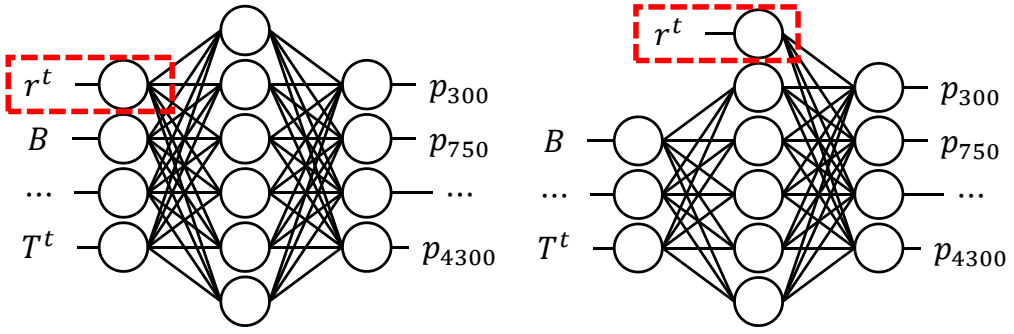
Figure 5.6: QoE ratio of Metis on different ABR algorithms and QoE metrics.

5.5.2 Performance Maintenance

We demonstrate the performance maintenance of Metis by comparing the QoE of original algorithms and decision trees converted with Metis. We thus measure the ratio of QoE by the Metis-generated decision trees and the original algorithms. A QoE ratio of less than 100% indicates a performance degradation. Since the QoE spans across positive and negative values, we normalize all the QoE values into a distribution with mean value as 1 and standard deviation as 1. We first measure the average normalized QoE and average QoE ratio across three types of QoE metrics and all traces, as shown in Figure 5.6. The average performance degradation is less than 3% for three algorithms (average QoE ratio of Pensieve is 97% in Figure 5.6), which is negligible compared to the performance improvement achieved by new algorithms.

5.5.3 Guide for Model Design

We present a use case to demonstrate that the interpretations of Metis can help the design of the DNN structure of Pensieve. As interpreted in §5.5.1, Metis finds that Pensieve sig-



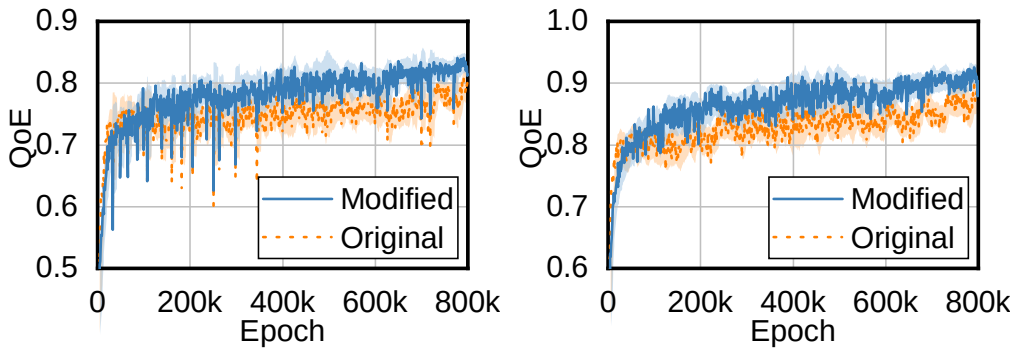
(a) Original structure. (b) Modified structure.

Figure 5.7: We modify the DNN structure of Pensieve based on the interpretations in §5.5.1. Although two structures are equivalent for the expressive ability, putting significant inputs near to the output will make the DNN optimize easier and better.

nificantly relies on the last chunk bitrate (r^t) when making decisions. This indicates that r^t may contain important information to the optimization.

To utilize this observation, we modify the DNN structure of Pensieve to enlarge the influence of r^t on the output result. As shown in Figure 5.7(b), we directly concatenate the r^t to the output layer so that it can affect the prediction result more directly. Although the two DNN structures are mathematically equivalent, they will lead to different optimization performance and training efficiency due to the huge search space of DNNs [104]. After putting the significant feature nearer to the output layer (thus simplifying the relationship between the significant feature and results), the modified DNN will focus more on that significant feature.

We retrain the two DNN models on the same training and test sets and present the results in Figure 5.8. From the curves of the original model and the modified model, we can see that the modification in Figure 5.7 improves both the training speed and the final QoE. For example, on the test set, the modified DNN achieves 5.1% higher QoE on average than



(a) Training set. (b) Test set.

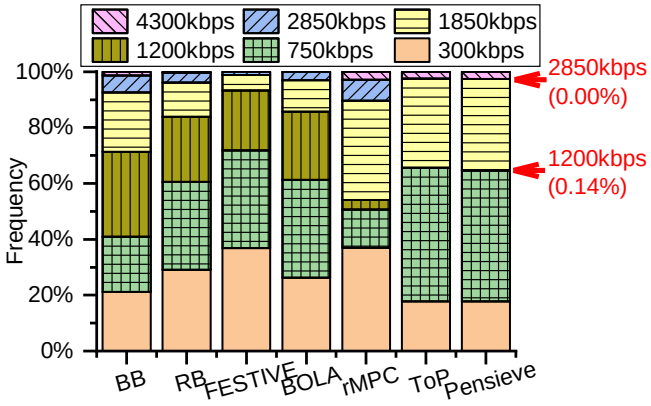
Figure 5.8: The modification in Figure 5.7 could improve both the QoE and the training efficiency. Shaded area spans $\pm$ std.

the original DNN[‡]. Considering the scale of views (millions of hours of video watched per day [256]) for video providers, even a small improvement in QoE is significant [180]. Moreover, the modified DNN can save 550k epochs on average to achieve the same QoE, which saves 23 hours on our testbed.

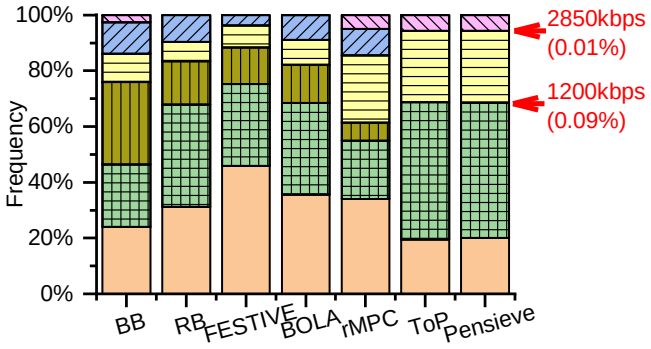
5.5.4 Enabling Debuggability

When interpreting Pensieve, as also reported in [94], we observe that some bitrates are rarely selected by Pensieve. The frequencies of selected bitrates of the experiments in §5.5.1 are presented in Figures 5.9(a) and 5.9(b). Among six bitrates from 300kbps to 4300kbps, two bitrates (1200kbps and 2850kbps) are rarely selected by Pensieve. The imbalance raises our interests since missing bitrates are *median* bitrates: the highest or lowest bitrates may not be selected due to network conditions, but not median ones.

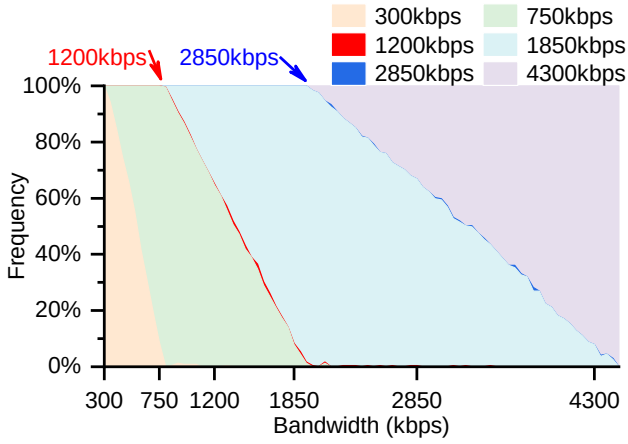
[‡]The offline optimality gap of Pensieve reported in [179] is 9.6%-14.3%.



(a) HSDPA traces (12250 actions).



(b) FCC traces (10045 actions).



(c) Fixed bandwidth with Pensieve.

Figure 5.9: For (a) and (b), Metis +Pensieve generates almost the same results with Pensieve, where 1200kbps and 2850kbps are rarely selected. (c) On a set of fixed-bandwidth links, 1200kbps and 2850kbps are still not preferred. Better viewed in color.

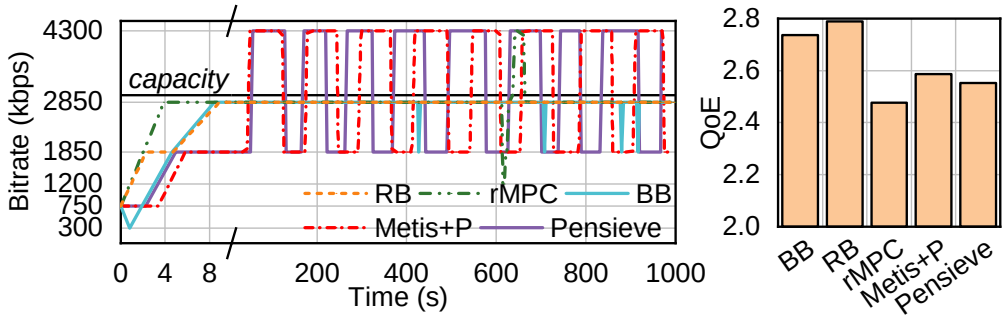


Figure 5.10: On a 3000kbps link, BB, RB, and rMPC learn the optimal policy and converge to 2850kbps. Metis +Pensieve (Metis +P) and Pensieve oscillate between 1850kbps and 4300kbps, degrading the QoE. Better viewed in color.

To further explore the reasons, we emulate Pensieve on a set of links with fixed bandwidth ranging from 300kbps to 4500kbps. As the sample video used by [179] is too short for illustration, we replace the test video with a video of 1000 seconds and keep all other configurations the same with the original experiment. As shown in Figure 5.9(c), 1200kbps and 2850kbps are still not preferred by Pensieve. For example, on a fixed 3000kbps[§] link, the optimal decision of which should always select 2850kbps. However, in this case, only 0.4% of selections made by Pensieve are 2850kbps, while the remaining decisions are divided between 1850kbps and 4300kbps. As shown in Figure 5.10, Pensieve oscillates between 1850kbps and 4300kbps, which is also mimicked by Metis +Pensieve. However, such a policy is sub-optimal. In contrast, other baselines learn the optimal selection policy and fix their decisions to 2850kbps, achieving a higher QoE. Similar observations can also be observed on a 1200kbps link (Appendix B.3).

Studying the raw outputs of Pensieve, we find that Pensieve does not have enough confidence in either choice and therefore oscillates between them. The probability of selecting

[§]The goodput (bitrate) in this case is roughly 2850kbps.

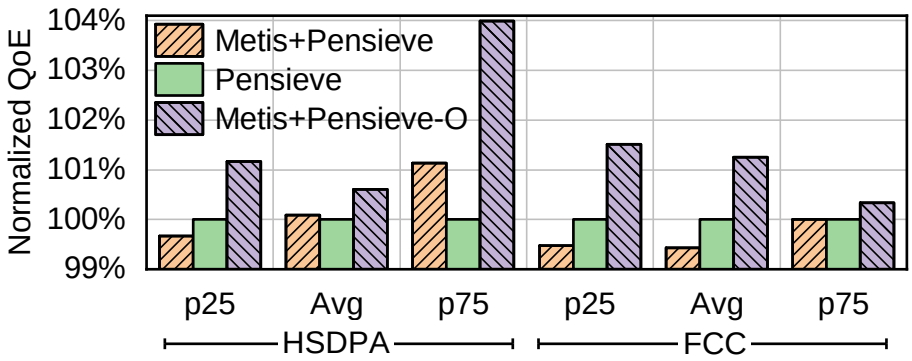


Figure 5.11: When converting DNNs to decision trees in Metis, oversampling the missing bitrates (Metis +Pensieve-O) improves the QoE by around 1% on average compared to the original DNN in Pensieve. QoE is normalized by Pensieve.

the optimal bitrate is at a surprisingly low level (Figure B.4 in Appendix B.3). The training mechanism of Pensieve may cause this problem. At each step, the agent tries to *reinforce* particular actions that lead to larger rewards. In this case, when the agent discovers that four out of six actions can achieve a relatively good reward, it will keep reinforcing this discovery by continuously selecting those actions and finally abandon the others. Making decisions with fewer actions brings higher confidence to the agent, but also makes the agent converge to a local optimum in this case.

Beyond discovering the problem as [94], Metis can also help fix the problem. Without Metis, since Pensieve is designed based on RL, network operators do not have an explicit dataset of bitrates. Network operators may have to penalize the imbalance of bitrate in the reward and retrain the DNN model for hours to days, without knowing whether the RL agent can learn to escape the local optimum itself. With Metis, the conversion from DNN to decision tree exposes an interface for network operators to debug the model. Since the dataset $\mathcal{D}$ to train the decision tree is highly *imbalanced*, as a straightforward solution, we oversample the missing bitrates to make sure their frequencies after sampling are around

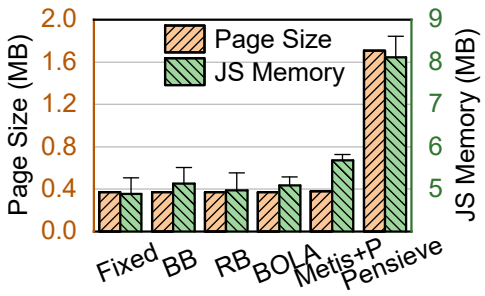


Figure 5.12: Compared to the original Pensieve model, Metis +Pensieve could reduce both page size and JS memory.

1%. As shown in Figure 5.11, the oversampled decision tree (Metis +Pensieve-O) outperforms DNNs by about 1% on average and 4% at the 75^{th} percentile on HSDPA traces.

5.5.5 Lightweight Deployment

Decision trees provided by Metis are also lightweight to deploy. We first demonstrate that the performance degradation between the decision tree and the original DNN is negligible (less than 2%). Therefore, directly deploying decision trees of Pensieve and AuTO online will reduce the resource consumption and bring further performance benefits.

Resource consumption. We evaluate the resource consumption (specifically, *page load time* and *memory consumption*) of Metis +Pensieve. To eliminate the influence of other modules in the DASH player, we compare these ABR algorithms with a `fixed` algorithm, which always selects the lowest bitrate.

For page load time, if the HTML page size is too large, users have to wait for a long time before the video starts to play. As shown in Figure 5.12, Fixed, BB, RB, and BOLA have almost the same page size because of their simple processing logic. Pensieve increases the page size by 1370KB since it needs to download the DNN model first. In contrast, Metis

+Pensieve has a similar page size with the heuristics. When the goodput is 1200kbps (the average bandwidth of Pensieve's evaluation traces), the *additional* page load time of ABR algorithms compared to `fixed` is reduced by 156×: Pensieve introduces an additional page load time of 9.36 seconds, while Metis +Penseve only adds 60ms.

We then measure the runtime memory and present the results in Figure 5.12. Due to the complexity of forward propagation in the neural networks, Pensieve consumes much more memory than other ABR algorithms. In contrast, the additional memory introduced by Metis +Pensieve is reduced by 4.0× on average and 6.6× on the peak, which is at the same level as other heuristics.

5.5.6 Metis Deep Dive

Finally, we overview the experiments that benchmark the interpretability of Metis. The detailed experimentation setup and more empirical results are deferred to the appendix.

Interpretation baselines comparison. We compare the performance of the decision tree in Metis against two baselines in the DL community. We implement LIME [217], one of the most typical blackbox interpretation methods in the DL community, and LEMNA [128], an interpretation method specifically designed for time sequences in RNN. We measure the misprediction rate and errors of three interpretation methods. The misprediction rates on two systems with Metis-based methods are reduced by 1.2×-1.7× compared to two baselines. Experiments are presented in Appendix B.4 in detail. The decision tree outperforms the other two interpretation methods, which confirms our design choice in §5.3.1.

Sensitivity analysis. We test the robustness of hyperparameters of Metis in Appendix B.5. For decision tree interpretations, we test the robustness of the number of leaf nodes. Results show that a wide range of settings (from 10 to 5000) perform well for Pensieve (accuracy variations within 10%).

Computation overhead. In Appendix B.6, our evaluation shows that converting fine-tuned DNNs into decision trees for Pensieve takes less than 40 seconds under different settings.

5.6 Discussion

In this section, we discuss some design choices, the generalization ability, limitations, and potential future directions of Metis.

Why not directly train a decision tree? As shown in §5.5.5, converted decision trees exhibit comparable performance to larger models. However, *directly* training the simpler model from scratch is difficult to achieve the same performance. We hypothesize that the first reason is that decision trees are non-parametric models, which are not designed for continuously parameter updating and structure adjusting. Even with recent advances in decision tree adjusting [178], the efficient adjustment relies on massive amount of training data. Another possible explanation behind this phenomenon is the *lottery ticket hypothesis* [120, 270]: training deep models is analogous to winning the lottery by buying a very large number of tickets (i.e., building a large neural network). However, we cannot know the winning ticket configuration in advance. Therefore, directly training a simpler model is

similar to buying one lottery ticket only, which has little chance to achieve satisfying performance.

Can Metis interpret *all* types of networking systems? Admittedly, Metis cannot interpret all DL-based networking systems. For example, network intrusion detection systems (NIDSes) are used to detect malicious packets with regular expression matching on the packet payload [198]. Prior DL-based methods introduced RNN to improve the performance of NIDSes [268]. However, since RNN (and other DNNs with recurrent structures) fundamentally contains *implicit* memory units, decision trees cannot faithfully capture the policy with only *explicit* decision variables. In the future, we aim to combine Metis with recurrent units, e.g., employing recurrent decision trees [80].

How to interpret deeper DNNs? Although our evaluation shows satisfying performance on three DL-based networking systems, compare to the applications of DNNs in other communities (Figure 5.2), those in networking systems are still at a preliminary stage: both Pensieve and AuTO have less than 10 hidden layers. Whether current approaches could scale to network systems with more complicated neural networks remains unknown. Nonetheless, on one hand, Metis might be scalable to deeper neural networks because deeper neural networks (regardless of training difficulty) sometimes have the same level of expressiveness compared to shallower ones [50, 131]. On the other hand, as a preliminary attempt, we adopt the traditional CART algorithm in decision tree training. More optimized decision tree representations [202] with tree-based regularization [262] during the training process of DNNs might interpret the policies more faithfully.

Will the generalization ability of DNNs be impaired? Although the generalization ability of DNNs is still under exploration, it is indisputable that the generalization ability of DNNs roots in the massive amount of parameters [206]. Despite that Metis performs well in our experiment settings as demonstrated in §5.5, the generalization ability of interpretations still needs investigation. There are two ways to further address the generalization ability of interpretations on different traces. On one hand, researchers can analyze the theoretic performance bounds of the interpretation [185]. On the other hand, network operators can deploy the interpretation results into the production environments and evaluate the online performance. We call on the community to devote more research efforts in this direction.

Will interpretations always be correct? Metis is designed to offer a sense of confidence by helping network operators understand (and further troubleshoot) DL-based networking systems. However, the interpretations themselves can also make mistakes. In fact, researchers have recently discovered attacks against the interpreting systems for image classification [134, 281]. Nonetheless, interpretations from our experiments are empirically sane (§5.5). Since the interpretations are concise and well understood, human operators could easily spot the rare case of erroneous interpretation.

5.7 Summary

In this paper, we propose Metis, a new framework to interpret DL-based adaptive video streaming systems. We apply Metis over a typical DL-based adaptive video streaming systems. Evaluation results show that Metis-based systems can interpret the behaviors of DL-

based networking systems with high quality. Further use cases demonstrate that Metis could help network operators design, debug, and deploy DL-based networking systems.

6

Application Layer on Data Path: Adaptive Frame-Rate

6.1 Introduction

Emerging network technologies like 5G have gotten both academia and industry excited about high-quality real-time communication (RTC) applications with ultra-high definition

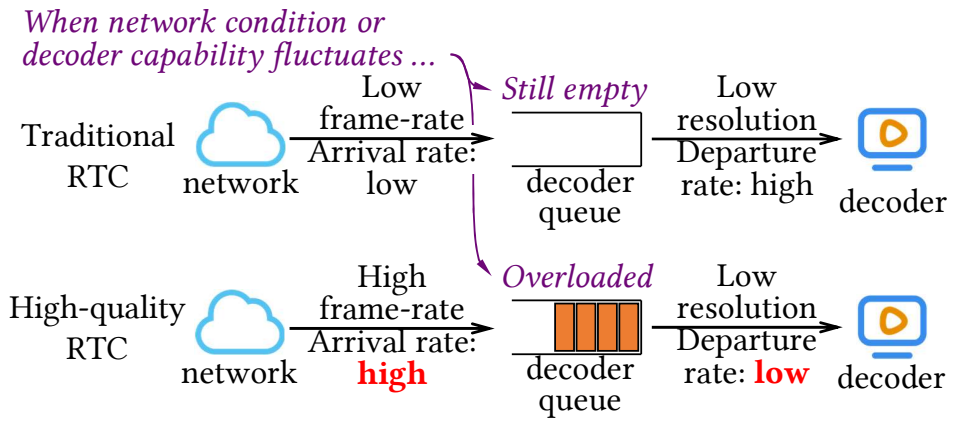


Figure 6.1: Comparison of the decoder queue between traditional and high-quality RTC applications. Due to the high frame rate and resolution, when network condition or decoder capability fluctuates, high-quality RTC applications may overload decoder queues, leading to high delay at the tail.

(UHD), high frame rate (HFR), and reduced delays. Examples include cloud gaming [140, 272], virtual reality [126, 213, 284] and 4K video conferencing [139, 163]. Some high-quality RTC services have already been deployed in production (e.g., cloud gaming from Google [15], Microsoft [19], Nvidia [12]). For example, the market share of cloud gaming reached one billion dollars in 2020, with an expected growth rate of 30% [63].

To achieve a satisfactory user experience, those applications need to stream with high resolution, high frame rate, and a low delay (§6.2). For example, cloud gaming services deliver content with a resolution of $\geq$1080p [15] and frame-rate of 60fps [207], while requiring a tail end-to-end delay of less than 100ms [143]. Streaming like this significantly improves users' experience and enables new applications.

This paper argues that, in addition to modulating bitrate to match network capacity, a high-quality RTC system must regulate the queuing at the decoder queue. For traditional standard quality RTC, the time required to decode a frame is much shorter than the in-

terarrival time of frames. Thus, the decoder queue is not a bottleneck and a traditional RTC service only needs to adjust the bitrate to match the network bandwidth. However, in high-quality RTC, the high frame rate reduces the time between the arrival of frames at the client, while the high resolution increases the decoding delay for each frame. At the decoder queue, the frame arrival rate frequently exceeds the departure rate, leading to a long queue, as shown in Figure 6.1. The video delivery is required to not only adapt the bit-rate to the network bandwidth but also coordinate with the decoder queue capacity. From measurements of our production cloud gaming service, Tencent Start [16], we find that video delivery without coordinating the queue capacity could introduce a non-negligible queuing delay at the client-side decoder queue. Moreover, such a queuing delay accounts for a large proportion of delayed frames in satisfying the much tighter delay requirement of high-quality RTC, especially when the network delay has been reduced with recent infrastructure developments (e.g., edge computing [197]). According to our measurements, among all frames with a total round-trip delay of $>$100ms, 57% of them have been delayed at the decoder queue for $>$50ms (§6.3.1). Our survey finds that the future demands of UHD and HFR video will further exacerbate the problem, even with the evolution of decoding hardware (§6.3.1). Therefore, for high-quality RTC, to reduce the end-to-end delay, it is essential to reduce the queuing delay at the decoder.

Not all interventions are effective at regulating the queuing at the decoder queue (§6.3.2). For instance, decoding delay is not affected much by bitrate. It is affected by resolution, but adjusting the resolution requires the client to request a new key frame. This consumes bandwidth and incurs several extra frame intervals of delay. Discarding a frame at the client also requires a new key frame, which incurs the same cost. Hence, we introduce an *adaptive*

frame-rate (AFR) controller, which controls the frame rate at the *encoder*. Reducing frame rate gives the decoder more time to process frames. Hence, it is effective at reducing the queue length. Further, edge streaming services offer short RTTs, which means the control loop to adjust the encoder's frame rate is short.

Note, there have been previous efforts to adapt the frame-rate (e.g., CU-SeeMe [130] decades ago). However, the development of decoding hardware had made it redundant in the recent decade, and traditional RTC in the recent decade is mostly bottlenecked in the network. In this paper, we show how high-quality RTC, with UHD resolution, HFR, and stringent delay requirements, has changed this. We further improve upon these proposals in two ways. First, existing control mechanisms are based on delay or queue length [119, 205, 260], which are slow to react since they need to wait for the queue to build up. AFR instead relies on the arrival and service processes in addition to the queue length to adjust the frame rate. Second, not all increases in decode queuing delay need to reduce the frame rate. For instance, when queuing delay increases due to a transient burst of arriving packets. Hence, AFR uses two control loops that adjust the frame rate at different time scales.

We implement the AFR controller on both simulators and the production of the cloud gaming service from Tencent Start [16]. Trace-driven simulations and deployments in the wild demonstrate that AFR could effectively reduce the tail queuing delay by up to 7.4×, and consequently reduce the ratio of frame stutters measured by total delay by up to 2.2× (§6.6.1 and §6.6.5) with negligible overhead. AFR has been deployed on Tencent Start since February 2021, serving millions of sessions. We will release the collected traces and the simulation code.

We make the following contributions:

- We carry out a month-long measurement campaign to motivate the significance of controlling queuing delay at the decoder queue, and identify the unique challenges from high-quality RTC with stringent delay requirements (§6.3).
- We design a hierarchical frame-rate controller, AFR, to control the decoder queue towards an ultra-short delay under different scenarios for high-quality RTC (§6.4).
- We evaluate AFR with both trace-driven simulations and large-scale deployments in production in the wild (§6.5). Our evaluation shows that both queuing delay and total end-to-end delay could be significantly improved (§6.6). AFR has been used in deployment for over one year.

6.2 Background: High-Quality RTC

High-quality RTC applications are attracting attention from the industry and academia. A series of high-quality RTC products have been released recently, including cloud gaming [12, 15, 19], virtual reality (VR) [20, 25, 32], and 4K videoconferencing [24]. For example, by generating high-quality content and streaming to the user via Internet, users can enjoy better video quality with low-cost devices. Specifically, the high-quality RTC has the following features standing out from traditional RTC applications:

- *High frame-rate.* Traditional RTC usually delivers content with a low frame rate (LFR) of 24fps [30]. However, high-quality RTC requires a frame rate of up to 60fps, some of which even require a frame-rate of 240fps [247].

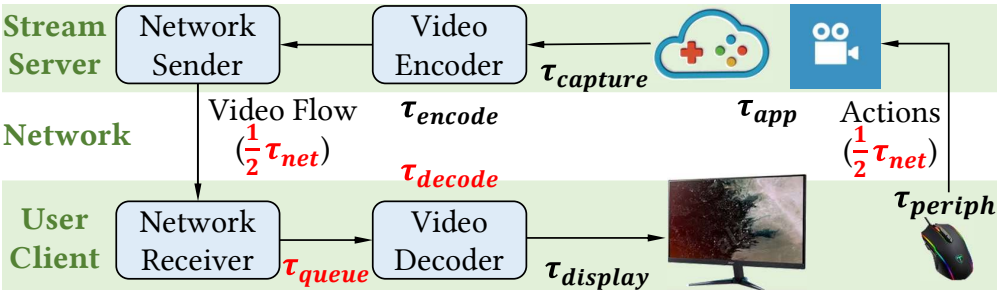


Figure 6.2: A general delivery pipeline of RTC services. We highlight the major contributing components in the tail end-to-end delay of high-quality RTC according to our measurements in red.

- *High resolution*. Most existing RTC applications are delivered at SD resolutions by default (e.g., 360p for Google Meet [23]). In contrast, high-quality RTC applications require a resolution of 1080p to 4K or higher [212].
- *Stringent delay requirement*. Furthermore, high-quality RTC applications also have stringent latency requirements. For example, videoconferencing requires a round-trip interaction delay of 150ms [30] and gaming for 100ms [143].

Existing delivery pipeline. To better understand the bottleneck of high-quality RTC, we present the key components of the existing RTC delivery pipeline in Figure 6.2. First, the video encoder captures the contents generated from video sources (e.g., gaming applications [73, 197]) and encodes them into video frames. Then, encoded frames are sent over the network from the streaming server to user clients. After that, on the client side, upon receiving new frames from the network, the decoder will decode those frames. Finally, decoded video frames will be displayed on users' displays.

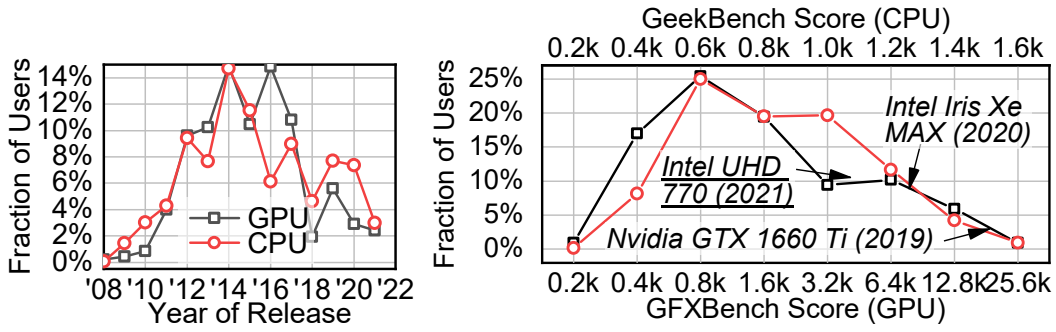
Optimization goal: low tail delay. With the intelligence from each community, the delay of each component has been intensively optimized in recent research efforts. To reduce the

network delay, existing providers either deploy stream servers at the edge [197, 250], introduce low-latency congestion controllers [47, 77], or suggest users use wired connections. For example, recent measurements unveil that cloud gaming services could deliver the RTC streams with an average round-trip network delay of 20ms [78, 197]. Similarly, streaming encoders are optimized for low latency to satisfy the stringent delay requirements in high-quality RTC services [119, 200, 234].

Meanwhile, optimizing the *tail* performance is also critical for user's experience for high-quality RTC [188]. The increase in tail delay will result in frame stuttering or freezing, degrading the user's experience. Quality of experience assessment frameworks in video streaming usually individually calculate the stuttering time as a penalty to the user's experience [98, 267]. Considering the high frame rate of high-quality RTC, further tails of 99th or 99.9th percentiles need to be focused on. For example, at the frame rate of 60fps, even the 99.9th percentile delay could happen every 16 seconds. Especially for applications such as cloud gaming, such a delay might lead to the loss of the game (e.g., stalls when the gamer is discovered by the opponent in a shooting game) [143, 227]. Therefore, it is essential to control the tail delay and reduce frame stutters for high-quality RTC.

6.3 Motivations and Challenges

In this section, we first explain the formulation of drastic queuing delay in high-quality RTC (§6.3.1). We then present our thinking over the design choice of adjusting frame rate (§6.3.2). We further analyze the unique challenges of effectively achieving an ultra-short queue (§6.3.3).



(a) Release date distribution

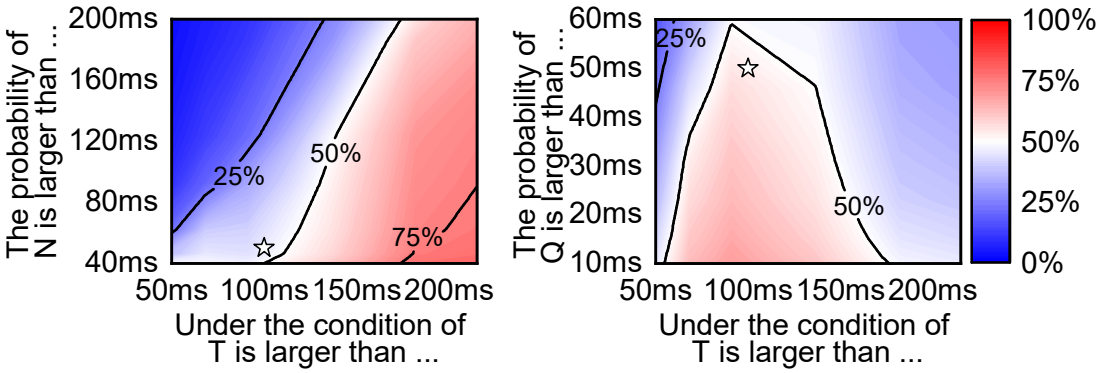
(b) Benchmark score distribution

Figure 6.3: Release year and benchmark score distribution of user devices in production. We use the single-core score in GeekBench [37] for the CPU benchmark and Aztec Ruins Normal Tier score in GFXBench [38] for the GPU benchmark.

6.3.1 Motivation: Drastic Queuing Delay

Observation: decoder queuing delay is a critical contributor to the total delay at the tail. We profile the delay of each frame at each stage in the delivery pipeline in Figure 6.2. We measure the Tencent Start cloud gaming service for a month in 2021, containing tens of thousands of users, with thousands of different CPU and GPU models. We present release dates and benchmark scores of CPU and GPU in Figure 6.3 and list top models in Appendix C.2.1. Unless other specified, all measurements in this paper are analyzed from this dataset.

According to our measurements, among all components in the pipeline, the network, queuing (at the decoder queue), and decoding delay are >10ms at the 99th percentile. We highlight them in red in Figure 6.2. The tail of the application and encoding delay is light since they are processed on commercial servers, which are stable compared to networks and



(a) Network delay.

(b) Queuing delay.

Figure 6.4: While network delay should usually be blamed when the total delay is above 200ms, queuing delay plays a dominant role among all frames with a total delay of more than 100ms. The color indicates the conditional probability $P(X > X_{th}|T > T_{th})$ for $X \in \{N, Q\}$. Stars denote X_{th}=50ms, T_{th}=100ms.

heterogeneous clients. Therefore, we focus on the network, queuing, and decoding delay in the following discussion. We leave the measurement results to Appendix C.2.2.

We investigate how these three components contribute to the increase of total delay at the tail. For each frame, we denote N, Q, D, and T as the network, queuing, decoding, and total end-to-end delay. We then calculate the conditional probability of $P(X > X_{th}|T > T_{th})$ for each $X \in \{Q, D, C\}$ from our measurements, where X_{th} and T_{th} are thresholds for statistics. A high conditional probability suggests that the component is more likely the cause of $T > T_{th}$. We calculate the conditional probability with different thresholds, and present the results for network delay and queuing delay in Figure 6.4.

As we can see, when analyzing the root causes of frames with T>200ms for traditional RTC services, network delay has a high probability (shaded red) to be blamed. However, when analyzing the frames with T>100ms, queuing delay dominates the increase of total delay. Our measurements show that among all frames with an end-to-end total delay

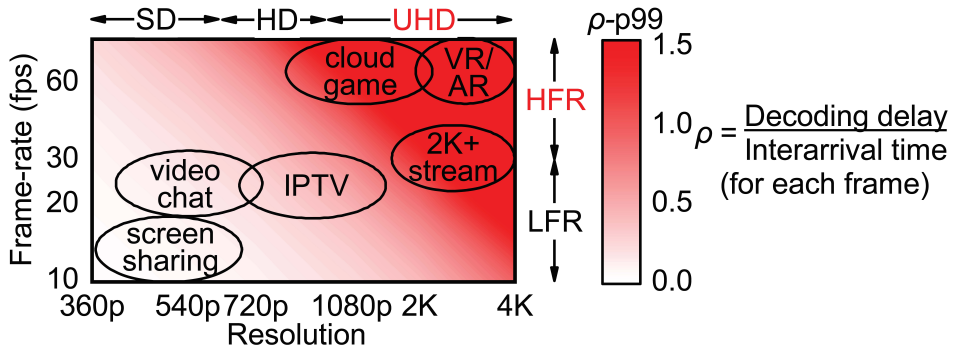


Figure 6.5: Illustration of the 99th percentile of the utilization ρ of the decoder queue. For high-quality RTC applications (in the top-right corner), the decoder queue is heavily loaded at the tail (shaded red), resulting in an increase of queuing delay at the tail.

of more than 100ms, queuing delay increase happens more frequently than all other component delays: 57% of them have a queuing delay of more than 50ms (stars in Figure 6.4). Considering the stringent delay requirement of ~100ms for high-quality RTC, the increase in queuing delay plays a dominant role.

Root cause: The UHD resolution and HFR jointly contribute to the increase in queuing delay. Compared to LFR streaming, HFR increases the arrival rate of the decoder queue by reducing the interarrival time between frames. Also, UHD decreases the departure rate compared to SD streaming by increasing the decoding delay of each frame.

Specifically, we illustrate how the frame rate and resolution could affect the load of the decoder queue by presenting the 99%ile queue utilization in Figure 6.5. We scale the distribution of interarrival time and decoding delay from our measurements to other frame rates and resolutions. As we can see, for traditional RTC services (the down-left corner), due to their low frame rates and resolutions, the decoder queue still has a utilization of $\rho \ll 1$

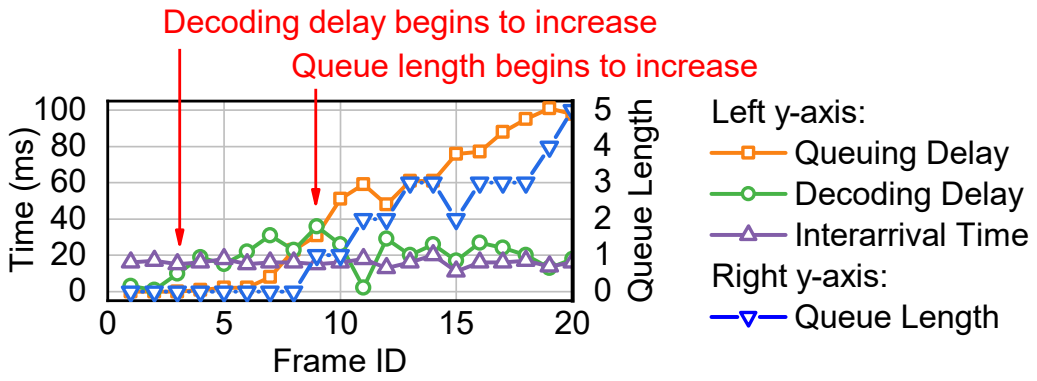
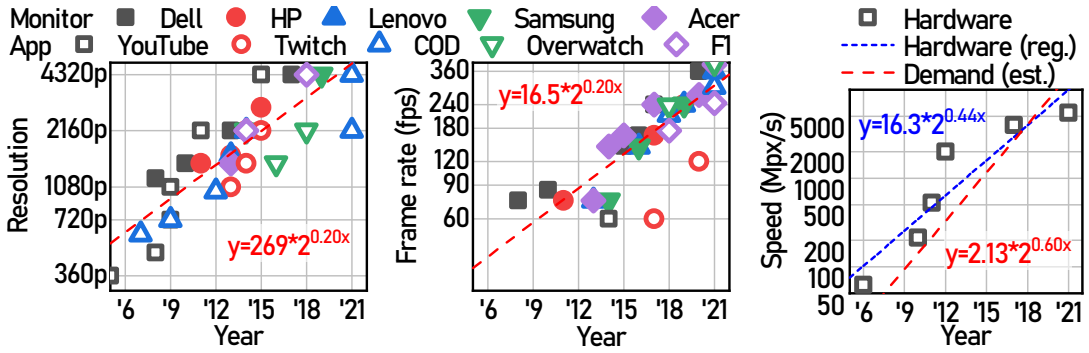


Figure 6.6: A trace for the accumulation of decoder queue. Note that this is an illustrative example – the distribution of all traces can be found in Appendix C.2.4.

at the tail. However, for high-quality RTC applications (the up-right corner), the decoder queue would be heavily loaded, leading to a drastic queuing delay.

The issue is the inconsistency of the decoder's performance *on average* and *at tail*. In fact, many of the hardware decoders that we measured claim to support UHD and HFR videos (e.g., Nvidia GTX series in Table C.2). However, according to our measurement, supporting UHD and HFR does not really mean *consistently* supporting. For example, the decoding delay can fluctuate due to numerous reasons including overheating at the client [214], CPU scheduling (§6.5.1), and the prediction errors [161], all of which are difficult to control for an application. From our measurement with devices in production, the decoding delay is 18ms at the 99th percentile even with hardware acceleration (Appendix C.2.2). Note that at the frame rate of 60fps, the interarrival time between frames is 16.7ms, resulting in a heavily loaded decoder queue at the tail.

We further analyze the necessity and sufficiency between the increase of other components and total delay in Appendix C.2.3 and figure out that the minor fluctuation of decoding delay leads to the increase of queueing delay. From the queuing theory, when the



(a) The maximum supported resolution and frame rate for the top 5 monitor vendors, two streaming platforms (YouTube and Twitch) and three games (Call-of-duty, Overwatch, and F1) [22].

(b) Decoding speed of existing hardware and required decoding speed from demands.

Figure 6.7: Decoding hardware cannot keep pace with the rapid increase of demands of videos with high resolution and frame rate. Note that the required decoding speed from demands is the frame rate times the *square* of resolution times the aspect ratio.

queue is heavily loaded, the queuing delay will drastically increase [97]. This is because while the decoding delay is continuously fluctuating, the queuing delay is accumulating all the fluctuations of precedent frames. Especially in heavy traffic, a minor fluctuation of the decoding delay could result in a magnitude increase in queuing delay. We refer the readers to [97] for more theoretical analysis. Illustratively, we present a trace from our production service in Figure 6.6. In the trace, the interarrival time is 16ms, and the decoding delay is 18ms, while the queuing delay is 54ms on average. The continual increase of the decoding delay, although not much by magnitude (18ms) and not long by duration (20 frames, approximately 0.3s), leads to a drastic queuing delay. If such a trace happens with a probability of 1%, we will have a 99th percentile decoding delay of 18ms, and a 99th percentile queuing delay of 55ms. In this case, the tail queuing delay is much higher than the decoding delay, which also contributes to more than half of the end-to-end stutters as analyzed in §6.3.1.

Trend: hardware decoders cannot keep pace with the increasing demands of UHD and HFR video. User demands for video have increased sharply, as shown in Figure 6.7(a). For example, the highest supported resolution and frame rate of YouTube have increased from 360p@30fps (7Mpx/s) in 2005 to 8K@60fps in 2015 (2Gpx/s), doubling every 14 months on average. Emerging services at 16K [212, 280] or 240fps [247] further indicate the future demands of UHD and HFR streaming.

However, the decoding speed of the hardware is not increasing as fast. We summarize the decoding speed of state-of-the-art video decoders from recent academic papers [88, 167, 280, 285, 286, 287]. As shown in Figure 6.7(b), the decoding speed of the state-of-the-art decoding hardware doubles only approximately every 27 months (blue dotted line). Meanwhile, we also calculate the required decoding speed from the existing demands of videos by multiplying the estimated resolution and frame rate from Figure 6.7(a) and plot the estimation in red in Figure 6.7(b). The required decoding speed from demands, doubling every 20 months, (red dashed line) increases much faster than the development of decoding hardware (blue dotted line), indicating the continuous incapability of decoding hardware for UHD and HFR videos.

In addition to the state-of-the-art hardware, there are still a considerable number of low-end and mid-end devices in our users. User devices, even in the same generation, could also be very heterogeneous. For example, in Figure 6.3, notice that the performance of Intel Iris Xe is 2× better than Intel UHD 770 even though the latter is more recent. Thus, there is heterogeneity in user devices even in the same generation. Moreover, new video codecs (e.g., H.265), although with a higher compression ratio, even slow down the decoding speed by up to 60% [71, 74, 174]. In this case, the mismatch between the decoder

and UHD and HFR videos will further exacerbate, making the queuing delay at the tail a lasting issue.

6.3.2 Choice: Controlling Proper Parameters

We motivate the need to adjust the frame rate. For an encoder, there are three parameters that could be independently set, including the frame rate, bit rate, and resolution. The encoder will automatically optimize other parameters (e.g., quantization parameters) based on current contents to achieve the target frame rate, bit rate, and resolution. We refer readers to [49] for more details on video codec.

We first analyze how these parameters could affect the delay of different components. When the bit-rate increases, the network delay will increase due to the congestion. When the resolution increases, since the decoder needs to decode frames with larger pixels, it needs a longer time to decode. The queuing delay depends on the enqueue rate (i.e., frame-rate) and the dequeue rate (i.e., decoding delay). In contrast, for example, if the bit-rate decreases, yet the resolution is kept the same, the decoding delay for each frame will hardly decrease due to the hardware design of the codec, which we further measure in Appendix C.2.4. Thus, relying on the total delay (e.g., Salsify [119]) would lead to ambiguity in taking effective actions to reduce the delay.

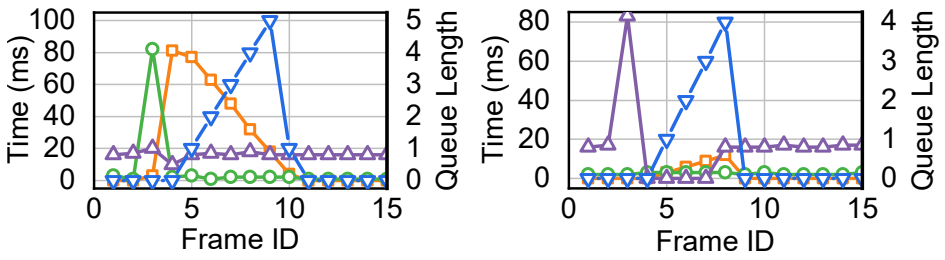
Therefore, we need to individually control respective parameters to reduce different delays. In response, we adjust the frame rate to control the queuing delay for high-quality RTC. When the fluctuations of the decoder and network result in an increase of queuing delay, it is essential to adjust the encoding parameters to reduce the queuing delay. In this case, after collecting measurements from the client and network, the encoder at the server

could accordingly adjust the frame rate for the following frames. We could dynamically specify certain timestamps where new frames are encoded.

We further discuss several potential solutions and concerns of adapting frame rates in Appendix C.1. In summary, adjusting the resolution or dropping frames is impractical due to the significant overhead of bandwidth. Statically choosing the frame rate based on the client model is also insufficient due to the fluctuation of decoding delay in the runtime. Moreover, since applications have limited control over users' systems, it is also impractical to control the user's system (e.g., pinning the application to a CPU core) for a large-scale production-level service [36]. In terms of frame-rate adaption, note that there are previous efforts in the adaption of frame-rate (e.g., CU-SeeMe [130] decades ago). However, as we discussed in §6.3.1, with the increase in resolution and frame-rate, and the stringent delay requirements, we need to reemphasize the significance of adapting frame rate now. We also show that it is timely enough to control the frame rate over the Internet.

6.3.3 Challenges

Achieving an ultra-short queue. To achieve an ultra-short queuing delay for the decoder queue, it is challenging to pick the appropriate indicator to inform the controller when it needs to take action. Existing signals (queue length [205] or queuing delay [119, 260]) fail to achieve an ultra-short queuing delay. Since the accumulation of the decoder queue is the consequence of the fluctuation of the arrival or departure process, both the queue length and queuing delay can only be observed when the queue has already been built up. For the example in Figure 6.6, while the decoding delay starts to increase at the 3rd frame, a non-



(a) Stalled decoder services. (b) Bursty network arrivals.

Figure 6.8: Two traces of transient fluctuations of the decoder queue from online traces. Legends are the same as Figure 6.6.

zero queue length can only be observed by the 9th frame. We also evaluate baselines based on queue length and queuing delay in §6.5.2.

In response, we want to capture the *earliest signal* to perceive the potential queuing delay. Therefore, instead of measuring the queuing delay, we want to estimate the potential increase of queuing delay predictively. For example, inspired by recent advances in congestion control [125, 164], a straightforward way is to measure the dequeue rate of the decoder queue to estimate the potential increase of the queuing delay.

However, in terms of tails, the arrival process is also fluctuating, which could also lead to an increase in queuing delays. For example, the network delay might increase by ten times at the 99th percentile than the median [77]. In response, to precisely avoid queue accumulation, we extend the designs of [125, 164]: AFR comprehensively measures the arrival and departure process and controls the queuing delay based on queuing theory. We introduce the design in §6.4.2, and evaluate the necessity of measuring the arrival process in §6.5.2.

Handling various events. Furthermore, the reason behind the formulation of the decoder queue in high-quality RTC is complex. As we introduced in §6.3.1, the stationary degra-

dation of decoding capacity could lead to the accumulation of the decoder queue, e.g., the traces in Figure 6.6. Besides, the decoder queue could also be accumulated due to transient contingencies. For example, from our experiences in production, the decoder might contingently experience a sudden decoding lag of ∼100 milliseconds (e.g., the 3rd frame in Figure 6.8(a)). The sudden interference in wireless channels might also lead to the bursty arrival of several frames (e.g., the 4th to 8th frames in Figure 6.8(b)). In both cases, the decoder queue will be accumulated. Since these transient fluctuations happen suddenly, it is challenging for the controller to react by measuring enqueue and dequeue rates.

Thus, AFR differentiates the causes of queue accumulation and reacts respectively to fluctuations at different time scales. We design a stationary controller to avoid queue accumulation in heavy traffic (§6.4.2), and a transient controller to reduce the queuing delay in contingencies (§6.4.3).

6.4 Design – Adaptive Frame-Rate (AFR)

We first analyze the overall workflow of AFR in §6.4.1, and then present the two controllers of AFR (§6.4.2, §6.4.3).

6.4.1 Workflow Overview

The workflow of AFR is presented in Algorithm 3. Specifically, the stationary controller (§6.4.2) maintains the queue around an ultra-short target based on dynamics of enqueue and dequeue processes. By measuring the statistics of both processes, AFR calculates the expectation of the queuing delay based on queuing theory. The frame rate can therefore be optimized towards a given queuing delay target (line 1). The transient controller observes

Algorithm 3: Hierarchical AFR control.

Input: Enqueue process $\{A_n\}$, dequeue process $\{S_n\}$, queue states Q. (A_n denotes the interarrival times, and S_n denotes the decoding delays of frames $\{n\}$.)
Output: Target frame rate f.
1 $f_0 = \text{StationaryController}(\{A_n\}, \{S_n\})$
2 $\alpha = \text{TransientController}(Q)$
3 $f = \alpha f_1$

the queue states Q (queue length and queuing delay) and calculates the discounting factor $\alpha \leqslant 1$ (line 2) to further decrease the frame rate when the queue formulates. The final frame-rate is the stationary frame-rate f_0 discounted by α (line 3). In this case, AFR can react to various scenarios of queue accumulation.

6.4.2 Stationary Controller

As introduced above, we measure the arrival and service processes and control the expected queuing delay of the queue. Specifically, we use the Kingman formula as an approximation of the expectation of queuing delay. Kingman formula is a widely adopted approximation formula of queuing delay [154] for G/G/1 queues. Compared to other approximation methods, in this paper, we adopt the Kingman formula to estimate the queuing delay since its estimation is from *both arrival and departure processes* without relying on queue states, which could provide the earliest signal for the potential queuing delay. According to the Kingman formula, the expectation of queuing delay τ_{queue} follows:

$$\mathbb{E}\left(\tau_{queue}\right) \approx \left(\frac{\rho}{1-\rho}\right)\left(\frac{c_a^2+c_s^2}{2}\right)\mu_s \tag{6.1}$$

where

$$c_a = \sigma_a/\mu_a, \quad c_s = \sigma_s/\mu_s, \quad \rho = \mu_a/\mu_s \tag{6.2}$$

(μ_a, σ_a) and (μ_s, σ_s) are the mean and standard deviation of the arrival and service processes:

$$\mu_a = \mathbb{E}\{A_n\}, \sigma_a = \sqrt{var(A_n)}, \mu_s = \mathbb{E}\{S_n\}, \sigma_s = \sqrt{var(S_n)} \tag{6.3}$$

From Eq. 6.1, the queuing delay is related to the following factors:

- Queue utilization ρ. The queuing delay will increase when the queue is overloaded ($\rho \to 1$). The current frame rate and decoding delay determine the queue utilization.
- Arrival and service fluctuations c_a and c_s. When the arrival or the service processes fluctuate, the queuing delay will also increase.
- Service time μ_s. Finally, the queuing delay scales with the average decoding delay.

Therefore, we control the expected queuing delay by controlling the right-hand side (RHS) of Eq. 6.1. We set $\mathbb{E}\{\tau_{queue}\}$ to a pre-defined queuing delay target W_0. Consequently, the target frame-rate f_0 could be calculated as:

$$f_0 = \rho/\mu_s = 1/\left(\mu_s \cdot \left(1 + \frac{\mu_s}{W_0} \cdot \frac{c_a^2 + c_s^2}{2}\right)\right) \tag{6.4}$$

Discussion: Approximation method. The AFR mechanism supports any approximation formula by design. There are other research efforts to control the queue. For example, recent efforts in congestion control [125, 164] directly set the target utilization (e.g., setting $\rho = 0.95$) and calculate the enqueue rate. In this paper, we adopt Kingman formula to capture *both the arrival and departure processes*, as discussed in §6.3.3. We also evaluate the performance of other baselines in §6.6.1.

Measurements of queuing dynamics. According to Eq. 6.4, we need to measure the mean and variance of the arrival and service processes. Similar to the RTT measurements

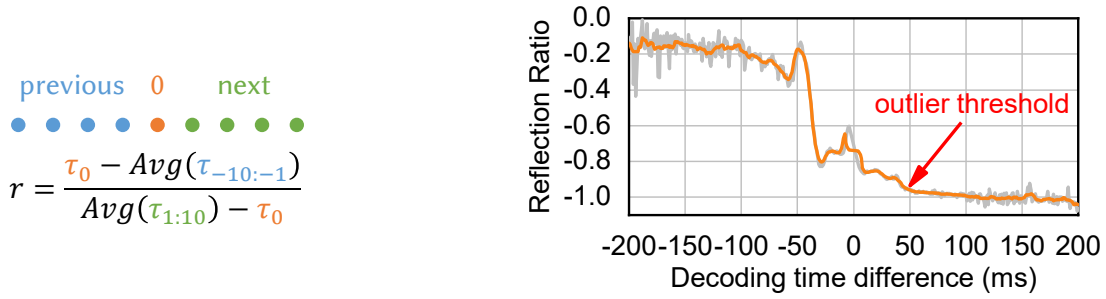
in TCP [145], we adopt the exponentially weighted moving average (EWMA) and exponentially weighted moving variance (EWMV) to estimate the $\mu_s, \sigma_s, \mu_a, \sigma_a$ in Eq. 6.1 and 6.2.

$$\begin{aligned} \hat{\mu}_n &= \xi_\mu x_n + \left(1 - \xi_\mu\right) \hat{\mu}_{n-1} \\ \hat{\sigma}_n &= \sqrt{\xi_\sigma \left(x_n - \hat{\mu}_n\right)^2 + \left(1 - \xi_\sigma\right) \hat{\sigma}^2_{n-1}} \end{aligned} \tag{6.5}$$

where x_n denotes interarrival time A_n or service time S_n. $\hat{\mu}_n$ and $\hat{\sigma}_n$ are the EWMA and EWMV. ξ_μ and ξ_σ are the discounting factors for the measurement of mean and standard deviation, trading off between precision and sensitivity.

However, due to bursty arrival or stalled services (§6.4.1), both the arrival and service processes could have significantly deviated value. For example, the 3$^{\text{rd}}$ frame in Figure 6.8(a) has a decoding time of 82ms while other frames are below 4ms. Such outliers will significantly deviate the estimation of stationary statistics for a long period. In fact, as we discussed in §6.4.1, these contingent events are designed to be handled by the transient controller. Therefore, we need to filter those outliers out to precisely estimate the stationary status of arrival and service processes. Due to the highly skewed distribution of decoding delay, existing outlier removal mechanisms based on standard deviation (e.g., the three-σ rule [215, 228]) suffer from differentiating stationary state transitions from outliers.

To capture the transitions of the status of decoders while eliminating the influence of the contingent outliers, we introduce an outlier removal mechanism based on priori knowledge from measurements in production. The key intuition is that *decoding delay differences* ($S_n - S_{n-1}$) are related to the probability of being outliers. For example, an increase of 20ms on decoding delay is probably the transition between stationary states (Figure 6.6). However, a sudden increase of 80ms on decoding delay is likely to indicate that decoding delay is



(a) Illustration

(b) Measurements in production

Figure 6.9: Reflection in outlier removal. Figure 6.9(b) presents the frequency of frames with $r \in [\frac{1}{C}, C]$. Measurement details in §6.5.2.

an outlier, which is usually the scenario of contingent stalls in Figure 6.8(a). This is because commercial decoders are usually able to decode frames at the frame rate of 24fps on average. According to our measurements, when the decoding delay difference is above 50ms, the possibility of being an outlier for that frame is 95%. Thus, we remove frames with a decoding delay difference of >50ms in the stationary controller, and leave the control of those frames to the transient controller.

We further characterize our observation based on measurements in production. As shown in Figure 6.9(a), we quantify the outlier with *reflection ratio r*, which illustrates the recovery of decoding delay before and after the potential outlier. The numerator is the difference between the current decoding delay (τ_0) and the average decoding delay of the previous 10 frames ($\tau_{-10:-1}$), and the denominator is the difference between τ_0 and future decoding delay. For outliers of contingent stalled service (e.g., the 3rd frame in Figure 6.8(a)), their reflection ratios would approach -1. This is because previous frames and subsequent frames have similar decoding delays, while the outlier has a much higher decoding delay ($\tau_0 \gg \tau_{-10:-1} \approx \tau_{1:10}$).

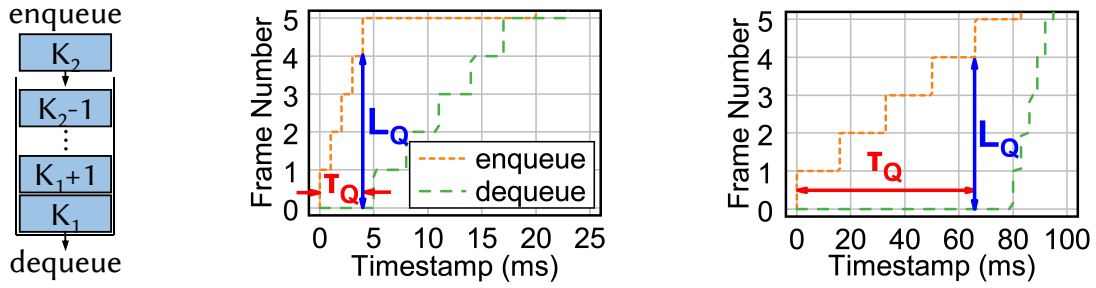
We then plot the relationship between the difference of decoding delay ($\tau_0 - \tau_{-1}$) and the average reflection ratio (r) of all frames with the same difference from our measurements in Figure 6.9(b). When the decoding time difference is larger than 50ms (marked with a red arrow), the average reflection ratio is less than -0.95, indicating that most frames in this scenario are outliers. Therefore, the stationary controller in AFR does not calculate the frames with a decoding delay difference larger than 50ms.

Convergence time analysis. To help operators to better understand the behavior of the stationary controller, we investigate the convergence of the stationary controller during state transitions of the service process. We want to answer the following question: During the transition from stationary state (μ_1, σ_1) to (μ_2, σ_2), how long will the stationary controller take to converge to the new frame-rate and drain the potential accumulation of the queue due to the transition?

We outline the main conclusion here and leave the detailed analysis in Appendix C.5. When the control loop (round-trip delay) of AFR is τ frames, the convergence time T_0 is bounded w.r.t. τ and W_0, and is acceptable for most scenarios. For example, when the average control loop of AFR is the interarrival time of one frame (τ=1), and W_0=2ms, the stationary controller could converge to the new stationary state within 2 frames. We illustrate the convergence time of the stationary controllers with more settings in Appendix C.5.

6.4.3 Transient Controller

The transient controller is designed to handle the contingent queue accumulations (§6.4.1). Therefore, we need to first understand how we should react to these queue contingencies.



(a) Queue. (b) Bursty network arrival. (c) Stalled decoder service.

Figure 6.10: Differences between bursty network arrivals and stalled decoder services. The y-axis is the accumulated enqueue/dequeue frames. For example, the enqueue curve in Figure 6.10(b) increases from 1 to 2 at 1ms, indicating that frame #2 enqueues at 1ms.

Understanding queue contingencies. As shown in Figure 6.8(a) and 6.8(b), both stalled decoder services and bursty network arrivals will cause a sudden increase in queue length. We illustrate the enqueue and dequeue events of two contingencies in Figure 6.10. In Figure 6.10(b), 5 frames arrive at the client together within 4ms, resulting in a queue length of 4 when the 5th frame arrives and observes, as illustrated with the L_Q (blue arrow). In Figure 6.10(c), the decoder takes 80ms to decode the 0th frame, when queued frames cannot be dequeued to the decoder. Therefore, upon the arrival of the 5th frame, it also observes a queue length of 4.

However, the bursty network arrivals and stalled decoder services should be handled separately. In the scenario of bursty network arrivals, the bottleneck of total delay is still in the network due to its long network delay. As long as the decoder is functional, even if multiple frames arrive at the queue simultaneously, they could be processed efficiently (Figure 6.8(b)). In this case, the queue will be drained in a short time, and we do not need to reduce the frame rate. In contrast, the stalled decoder service will drastically increase the

queuing delay of subsequent frames and needs adaption (Figure 6.8(a)). Thus, we need to differentiate between the two scenarios.

Since both scenarios result in an increase in queue length, they cannot be effectively differentiated with queue length only. Our insight is that we can differentiate them with the sojourn time of the first frame in the queue. As shown in Figure 6.10(a), at the arrival of frame K_2, the sojourn time τ_Q of the first frame K_1 and queue length L_Q observed by K_2 are:

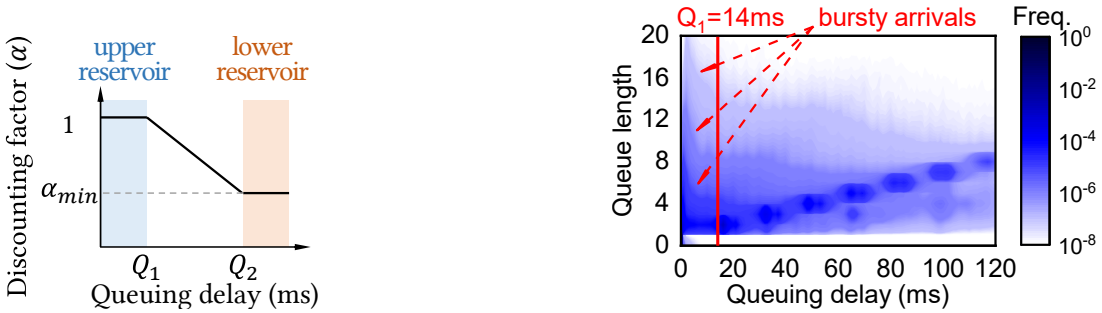
$$\tau_Q = t_{enq}^{(K_2)} - t_{enq}^{(K_1)}, \qquad L_Q = K_2 - K_1 \tag{6.6}$$

where $t_{enq}^{(i)}$ is the enqueue timestamp of frame #i, and frame #K_1 is the frame at the head of the queue. For bursty network arrivals, since frames arrive at the decoder queue simultaneously, when the last frame of the burst arrives, the first frame has only been queued for a short time. For example, τ_Q in Figure 6.10(b) is 4ms (marked red). In contrast, for stalled decoder service, the head frame has been blocked for a long time, leading to a high τ_Q of 66ms in Figure 6.10(c). Therefore, we use τ_Q to adjust the frame rate in the transient controller.

Feedback control. For the transient controller, the design space is to find out a mapping between the discounting factor α and the queuing delay τ_Q. Since the transient controller is designed to reduce the frame rate based on the results of the stationary controller, the possible range of α satisfies:

$$f_{min}/f_{max} = \alpha_{min} \leqslant \alpha \leqslant 1 \tag{6.7}$$

where f_{min} and f_{max} are the lower and upper bounds for frame rate required by the application. Since longer τ_Q indicates a more severe load of the queue, the discounting factor



(a) α-τ_Q mapping.

(b) L_Q-τ_Q frequencies.

Figure 6.11: Illustrations and measurements of the transient controller. A series of linearly distributed dark blue clusters in Figure 6.11(b) indicate that L_Q and τ_Q are linearly correlated.

should decrease with the increase of τ_Q. Besides, the α-τ_Q mapping should also have the following properties:

First, avoid overreactions. As we discussed above, for bursty network arrivals, τ_Q will also slightly increase due to the volumetric arrived frames. However, since such a transient queue accumulation will be cleared quickly as long as the decoder is functional (Figure 6.10(b)), we should not decrease the frame rate. Therefore, we need to introduce an upper reservoir (as shown in Figure 6.11(a)) to avoid overreactions. In the upper reservoir, when a non-zero but small τ_Q is observed ($0 \leqslant \tau_Q \leqslant Q_1$), the transient controller will not decrease the frame rate. The reservoir threshold Q_1 should be set based on measurements. We measure the observed L_Q and τ_Q from frames and present the results in Figure 6.11(b). Peaks near the left axis (marked by red dashed arrows) represent frames with a long L_Q yet with a short τ_Q, which are due to the bursty network arrivals. Therefore, we set Q_1 to filter out those bursty arrival-related peaks (e.g., Q_1=14ms in our deployment, the red line in Figure 6.11(b)).

Second, respond timely. Due to the stringent delay requirements of high-quality RTC applications, a long queuing delay will drastically degrade the users' experiences. Therefore, we need to control the slope of the mapping in Figure 6.11(a) to effectively reduce the queuing delay. Since α is lower bounded, we could control the slope of the mapping by introducing a *lower reservoir*, as shown in Figure 6.11(a). We set Q_2 as the maximum tolerable queuing delay:

$$Q_2 = \max\left(Q_1, Deadline - \tau_{network} - \tau_{decode}\right) \tag{6.8}$$

where $\tau_{network}$ is the round-trip network delay, and τ_{decode} is the decoding delay μ_s. *Deadline* is the requirement for the total delay of the application. Based on users' experiences in the human-machine interaction and our operational experiences, we set *Deadline* to 100ms in our deployments [143].

6.5 Implementation

We implement the AFR with a frame-level trace-driven simulator, and deploy the AFR onto a production high-quality RTC service in the wild. In this section, we present the design of our simulator (§6.5.1), introduce the simulation setup (§6.5.2) and the deployment setup (§6.5.3).

6.5.1 Simulator Design

To faithfully compare and replay the traces for different queue control algorithms, we design a simple simulation environment that models the dynamics of RTC. The simulator maintains the decoder queue and replays the traces collected from online services, where the traces contain the decoding delay, network delay, original queuing delay, and also the

	Category	Session	Frame	Playtime
(1)	Windows+Ethernet	29.7k	6.35 B	34.2k hours
(2)	Windows+WiFi	6.4k	1.12 B	6.2k hours
(3)	MacOS+Ethernet	0.4k	40.9 M	0.2k hours
(4)	MacOS+WiFi	2.1k	216 M	1.1k hours
	Total	38.1k	7.73 B	41.7k hours

Table 6.1: Distribution of our traces on the client type.

arrival timestamp for each frame. Specifically, frames arrive at the decoder queue according to timestamps in traces, wait in the decoder queue for dequeuing, and are decoded according to decoding delays in traces. To avoid frequently sending frame-rate adjustment requests to the servers, frame rates are quantized at the level of 5fps, which is also followed by our online deployment. We implement the potential interference from CPU time-slicing: since the fetching of frames to decoders depends on the CPU, there are possible cases where fetching the frame from the queue to the decoder needs waiting to be scheduled by the CPU by up to several milliseconds [68]. Therefore, we further profile such a delay in the traces and introduce the scheduling waiting time in our simulator. We also implement the response time of the encoder between the new frame-rate actions and new frames generated with the updated frame rate, according to our measurements in §6.6.4. Please refer to Appendix C.3 for implementation details.

6.5.2 Simulation Setup

Traces. We measure the frame-level statistics of our cloud gaming service (introduced in §6.3.1) on two types of clients (Windows and MacOS) and access networks (Ethernet and WiFi). We profile each step of received frames in one of our production clusters for 24 days

in December 2020. This results in a dataset with 7.73 billion frames and 41.7k hours of playtime (Table 6.1), which is the largest frame-level dataset for interactive streaming to the best of our knowledge.

Parameter settings. There are several parameters in AFR to be determined. Except for the parameters related to the transient controller (§6.4.3), we set W_0 in the stationary controller to 2ms and the discounting factors in EWMA $\xi_{arrv} = 0.033$ and $\xi_{serv} = 0.25$. We discuss the sensitivity of those settings and their influence on the performance in §6.6.3.

Metrics. In the evaluation, we mostly measure the delays (including the queueing delay and the end-to-end total delay). As we discussed in §6.2, the delay in interactive streaming is orthogonal to other video quality metrics (e.g., PSNR [14] or SSIM [258]). The delay, which represents interactivity, is the main optimization goal in this paper. We demonstrate that AFR has negligible degradation on the video quality in §6.6.4.

Baselines. To evaluate the performance of AFR, we implement existing frame control mechanisms as follows:

- `DropTail` is the frame control mechanism in WebRTC [205]. When frames overflow the queue, the client will first clear the queue, then request a new key frame, and finally drop all frames until the next key frame arrives. We set the queue capacity to 16 frames.

- `QLen-S` observes the current queue length, skips frames from the content generator before the encoder if the queue length is $\geqslant 1$, and resumes if the queue length is < 1.

- `QWait-S`. We migrate the frame control mechanisms from existing academic efforts in our simulator [119, 260], and replace the signal from total delay to queuing delay to better reduce the queuing delay. Since these baselines are not designed for stringent delay requirements of 100ms, we also finetune their parameters with our traces. `QWait-S` skips frames before the encoder if the queuing delay is $\geqslant$32ms, and resumes if the queuing delay is $<$4ms.

Besides, to evaluate the effectiveness of different components in AFR, we also different variants of AFR:

- `AFR-QLen`. We demonstrate the insufficiency of controlling the frame rate with queue states with a feedback algorithm based on current queue length: it observes the current queue length at the arrival of each frame, and maps the queue length of {0, 1+} to frame-rate {60, 24}fps.

- `AFR-QWait`. A feedback algorithm maps current queuing delay of {(0, 4), (4, 8), (8, 12), (12, ∞)}ms to frame-rate of {60, 48, 36, 24}fps. The parameters have also been finetuned with our traces.

- `AFR-TX`. To demonstrate the effectiveness of measuring both the arrival and service process, we further implement a dequeue rate-based algorithm. `AFR-TX` measures the dequeue rate and sets the target frame-rate with $\rho = 0.8$, where ρ has been tuned with our traces. The dequeue rate is the reciprocal of decoding delay.

- `AFR-Kingman`. Moreover, we individually evaluate the stationary controller of AFR to further illustrate the effectiveness of the transient controller.

- AFR. Finally, we put all optimizations in this paper (both the stationary and transient controller) together.

We present how we tune the parameters, and evaluate the trade-offs between frame rate and queuing delay in §6.6.3.

6.5.3 Deployment setup

We finally deploy AFR onto our cloud gaming service. The gaming service X employs the H.264 codec to increase the coverage of hardware decoding and adaption towards heterogeneous clients*, and customizes the codec performance for the optimization of gaming. Tencent Start currently supports 13 production-level games, including action-adventure, first-person shooter, and real-time strategy games. To optimize the network delay, the service is accelerated with multi-access edge computing similar to [197, 250, 282]: Users are split into tens of operation regions with a geographical diameter of hundreds of kilometers. Cloud gaming servers are deployed on clusters in each operation region, resulting in an average round-trip network delay of 15ms (Appendix C.2.2).

The frame-rate adaption algorithms are implemented on the client side. The AFR controller continuously measures the statistics of the decoder queue, and sends requests to edge servers to adjust the frame rate when necessary. The edge server then forwards the frame-rate adjustment requests to both the video encoder and the gaming application. New frames will be generated following the new inter-frame interval. We evaluate the response timeliness and overhead of video encoder and gaming application in §6.6.4.

*Hardware decoding has a shorter decoding delay than software decoding and supports higher frame rates. H.264 has a higher coverage of hardware decoding support compared to other advanced codecs [169].

6.6 Evaluation

We evaluate the AFR controller in the following aspects:

- **Delay improvements.** We present the performance improvements: The ratio of frames with long queuing delay and total delay of AFR has been improved by 2.1×-26× and 13%-2.2× against existing baselines (§6.6.1).
- **Frame-rate maintenance.** We then demonstrate that AFR introduces negligible impacts on the metrics related to frame-rate (§6.6.2).
- **Parameter sensitivity.** Our evaluation shows that parameters in AFR have a wide range of settings to gain performance improvements against finetuned baselines (§6.6.3).
- **Microbenchmarking.** We further demonstrate that the timeliness, overhead, and image quality of frame-rate adjustments are satisfactory for online deployment (§6.6.4).
- **Deployment in the wild.** Finally, we report the A/B test results and the deployment progress of AFR on our cloud gaming service online (§6.6.5).

6.6.1 Delay Improvements

We compare the queuing delay and the total delay of each frame with AFR and baseline algorithms in four sets of traces (Table 6.1). We measure the queuing delay in two dimensions: we present the 99th percentile queuing delay and the ratio of frames with a queuing delay >50ms in Figure 6.12. We first analyze the results of AFR against three existing

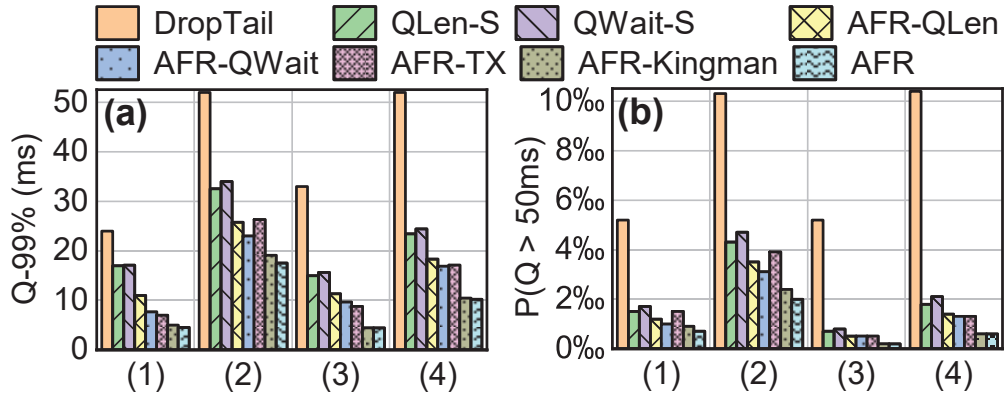


Figure 6.12: Simulation results of queuing delay (the 99%ile and the ratio of frames with >50ms queuing delay).

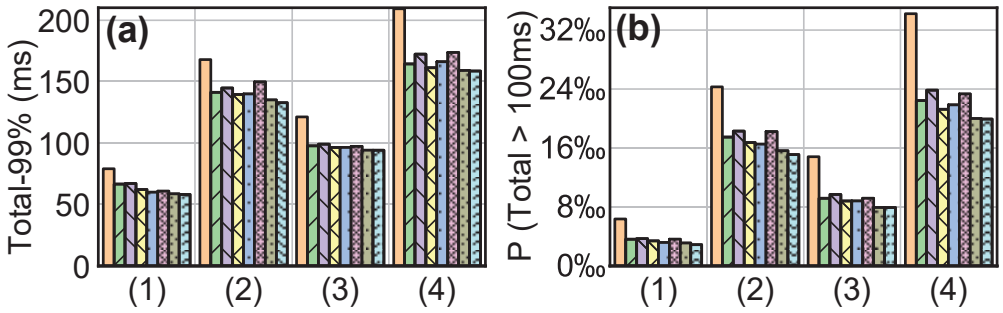
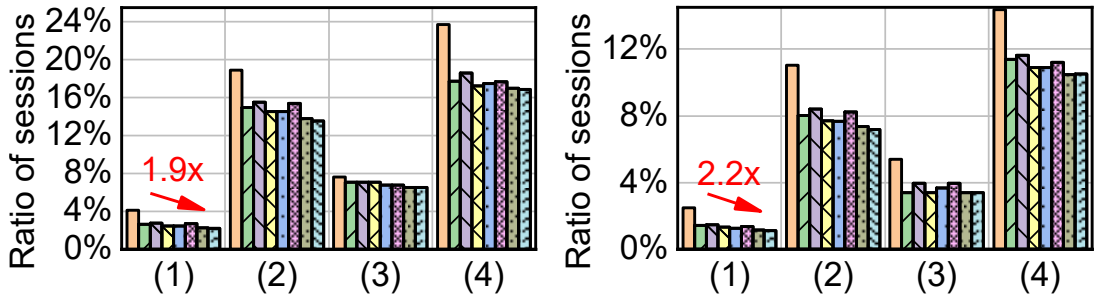


Figure 6.13: Simulation results of total delay (the 99%ile and the ratio of frames with >100ms total delay).



(a) Sessions with stutter ratio >5%.

(b) Sessions with stutter ratio >10%.

Figure 6.14: Ratio of sessions with different stuttered frames.

mechanisms (DropTail, QLen-S, and QWait-S). AFR could reduce the 99%ile queuing delay by 1.9× to 7.4×, and the ratio of severely queued frames by 2.1× to 26× on different sets of traces against three baselines. In this case, the 99%ile queuing delay could be squeezed to 6.9ms. This indicates that AFR could effectively achieve an ultra-short queuing delay. AFR also demonstrates satisfactory performance improvements on the *total end-to-end delay*, which is directly related to users' experiences. AFR improves the 99%ile total delay by 27% to 36%, and the ratio of severely delayed frames (total delay >100ms) by 1.6× to 2.2× in all traces. We also measure the *session stutter ratio*, i.e. the ratio of frames with a total delay of >100ms in a session, for each session. We then measure the ratio of sessions with a session stutter ratio of >5% and >10%, which indicates how many users suffer from unsatisfactory experiences and present the results in Figure 6.14. For the major population of our service (Cat. (1), Table 6.1), AFR reduces the stuttered sessions by 17% and 21% compared to the best of the three baselines. For other categories, the ratio of stutter sessions has also been reduced by 5% to 37%. AFR could significantly improve experiences for high-quality RTC.

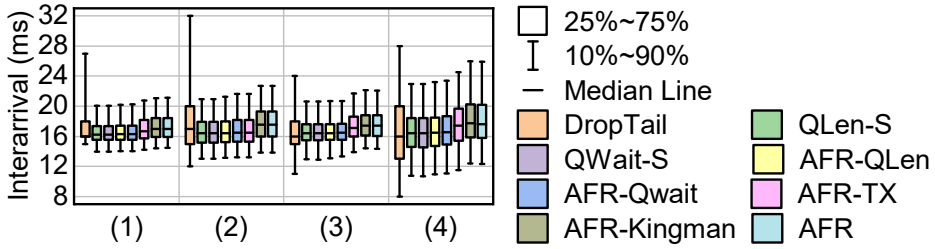
We further understand the performance improvements with the comparisons among different variants of AFR. Compared to DropTail, baselines based on queue states (AFR-QLen, AFR-QWait) could effectively reduce the queuing delay, indicating the necessity of actively controlling the queuing delay (§6.3.1). Compared to QLen-S and QWait-S, controlling the frame rate achieves better performance than skipping frames from the encoder. This is because skipping frames would drastically degrade the tail frame rate, for which the parameters of baselines are tuned (§6.6.3). AFR-TX could further reduce the queuing delay than the queue state-based baselines, indicating that observing the service process could know

the potential degradation in advance and effectively take actions, validating our analysis in §6.3.3. `AFR-Kingman` further improves the performance by 10% against `AFR-TX`, demonstrating that the fluctuating arrival of the high-quality RTC could also affect the estimation of the decoder queue. `AFR` finally reduces the tail queuing delay by 2-4% against `AFR-Kingman`, indicating the necessity of the transient controller to handle contingencies.

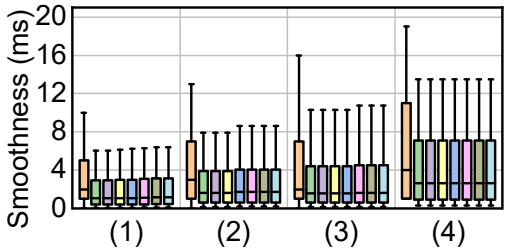
Besides, we also find that AFR has larger performance improvements when the network is better. The performance improvements on two sets of Ethernet traces (55% and 37% for Cat. (1) and (3)) are larger than the on WiFi traces (35% and 27% for Cat. (2) and Cat. (4)). Considering the ongoing deployment of next-generation access networks with better network conditions (e.g., 5G and WiFi 6), the necessity of controlling the decoder queue would be more significant.

6.6.2 Frame-rate Maintenance

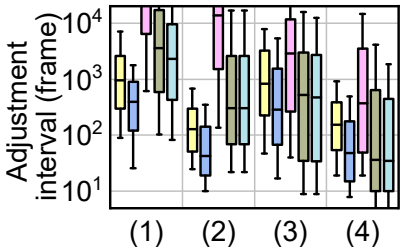
Besides, we also measure the effect of AFR on the frame rate. We first measure the interarrival time between frames at the arrival of each frame on the client. For example, a frame rate of 60fps should result in an interarrival time of around 16.7ms. We tune the parameters of each algorithm to keep the 99th percentile of their interarrival time at the same level (details in §6.6.3). Therefore, for 10-90th percentiles, as shown in Figure 6.15(a), most algorithms except for `DropTail` are comparable. Compared to the existing deployed mechanism `DropTail`, AFR even improves the tail user-perceived frame rate due to its better management of frame drops. AFR slightly decreases the median frame rate by 3%-9%, which brings the negligible quality of experience (QoE) degradation to users considering the improvements on delay [241, 271].



(a) Interarrival time between frames.



(b) Smoothness.



(c) Adjustment interval.

Figure 6.15: Frame-rate maintenance. Better viewed in color.

We further measure the smoothness of frame-rate, which might also have potential effects on users' experiences [98]. We measure the *differences of interarrival time* as an indicator of the smoothness of frame rate and present the results in Figure 6.15(b). Except for `DropTail`, all baselines and AFR have similar interarrival differences and are better than `DropTail`. This is mainly because that frame drops in `DropTail` will introduce a sudden increase of interarrival differences. Moreover, we also measure the frame adjustment interval and present the distributions in Figure 6.15(c). The median adjustment interval of AFR is hundreds to thousands of frames, which is much longer than the response time of frame-rate adjustment (§6.6.4).

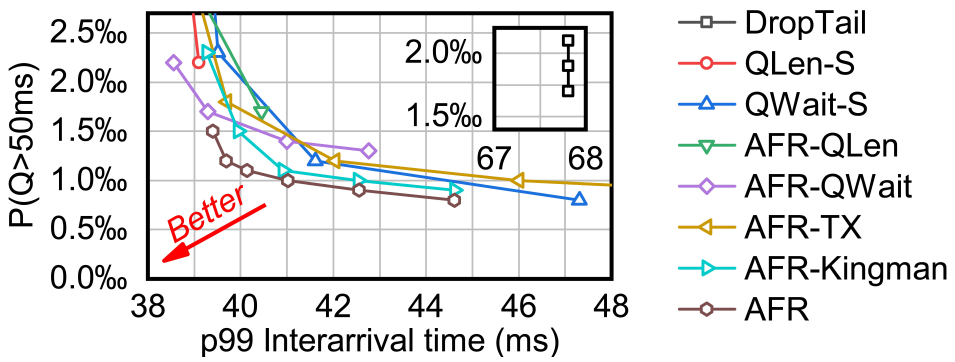
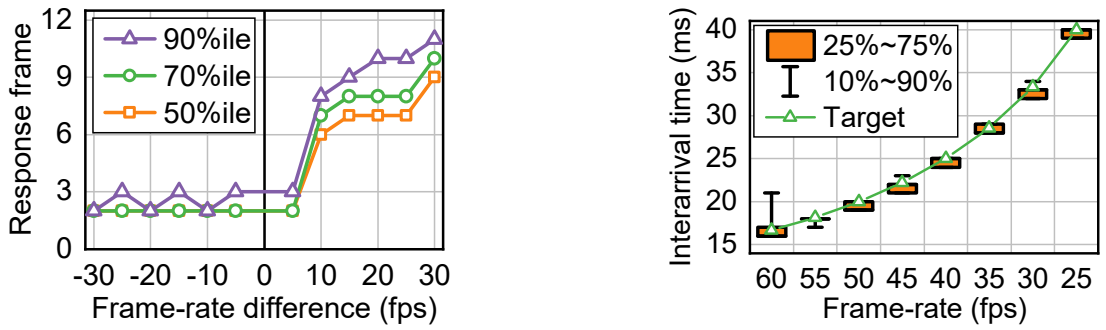


Figure 6.16: The trade-off between the tail interarrival time and queuing delay. We tune the parameters for baselines and AFR to illustrate the capability of each algorithm in the trade-off.

6.6.3 Parameter Sensitivity

We then evaluate the sensitivity of parameters in AFR and other baselines. We tune parameters of all baselines in §6.5.2: thresholds for skipping frames for `QLen-S` and `QWait-S`, mappings for `AFR-QLen` and `AFR-QWait`, ρ for `AFR-TX`, and W_0 for `AFR-Kingman` and AFR. We present the ratio of frames with queuing delay >50ms (P(Q>50ms)) and the 99th percentile of interarrival time on Cat. (1) traces in Figure 6.16. The down-left corner indicates the algorithm has a satisfactory trade-off between the queuing delay and the frame rate.

As we can see, AFR outperforms all other baselines in a wide range of settings, achieving a better trade-off between the queuing delay and frame rate. `QLen`-based algorithms are challenged in achieving ultra-short queuing delay: with the extremest parameters (skipping/decreasing frame-rate as long as queue length is non-zero), `QLen-S` and `AFR-QLen` could only achieve a P(Q>50ms) of 2.2‰ and 1.7‰, much higher than other baselines. This follows our analysis in §6.3.3 that queue length is too coarse-grained as a signal to control the queue with an ultra-short target. Meanwhile, skip-based algorithms could achieve lower queuing delay compared to frame-rate-based algorithms, yet with higher interarrival time.



(a) Adjustment timeliness.

(b) Frame-rate stability.

Figure 6.17: Effectiveness of frame-rate adjustment.

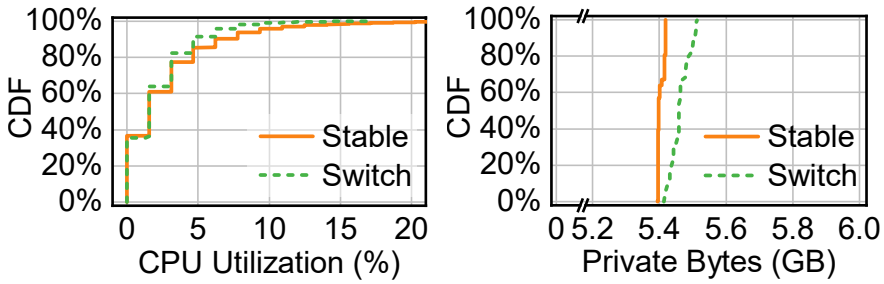
The parameters of all algorithms are tuned according to Figure 6.16 by aligning the 99th percentile interarrival time.

We also evaluate how different percentiles of queuing delay and total delay are affected by the setting of W_0 in Appendix C.4.3. The performance of AFR reacts sensitively to the setting of W_0, indicating that operators could effectively balance the total delay and frame rate by adjusting W_0. We further evaluate the sensitivity of the discounting factors ξ of the EWMA and EWMV in the transient controller (§6.4.3) in Appendix C.4.3, demonstrating how operators should set these parameters to balance between the precision and sensitivity.

6.6.4 Microbenchmarking

We also benchmark AFR in a testbed of our cloud gaming service.

Effectiveness of frame-rate adjustment. We first measure the responsiveness and precision of frame-rate adjustment at the video encoder. We enumerate all frame-rate switching within $\{25, 30, \cdots, 60\}$fps, and measure how many frames the encoder needs to take to steadily output video streams at the new frame rate. The response time measured by



(a) CPU utilization. (b) Memory utilization.

Figure 6.18: Frame-rate adjustment overhead.

the unit of frame (i.e. *response frame*) is presented in Figure 6.17(a). For each group of settings, we repeat the experiments 100 times to eliminate the randomness. When decreasing the frame rate, the 90%ile response frames is less than 3 frames, indicating the encoder and gaming application could decrease the frame-rate timely. This could effectively alleviate the overload of the decoder queue. When significantly increasing the frame rate, the frame rate might be slightly delayed to change. This is because the frame rate at the client side follows the bucket effect. Either encoder or the gaming application decreases the frame rate will lead to a decrease of the final frame rate, while the increase of frame rate needs an increase from both components. Even so, the tail response frame is <10 frames, which is much less than the adjustment interval (Figure 6.15(c)).

We then measure the fluctuation of the frame rate of the output of the streaming encoder. We set the frame rate to several levels as above, and measure the interarrival time between each frame. For each frame rate, we measure the interarrival time for 30,000 frames and present the distribution in Figure 6.17(b). The interarrival time between frames largely falls around the target frame rate. Therefore, unlike the fluctuating bit-rates in video streaming [141], frame-rate could be precisely controlled by the encoder.

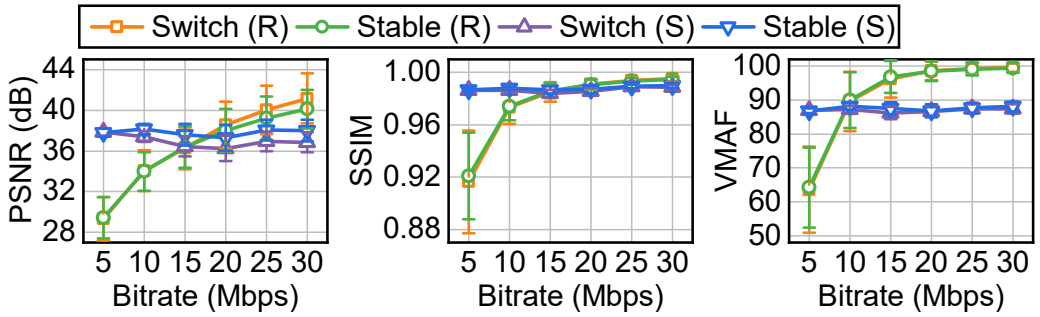


Figure 6.19: The image quality differences of AFR and the original video tested in a running scene (R) and stable scene (S). The error bar represents the standard deviation.

Frame-rate adjustment overhead. We further measure the potential processing overhead of frame-rate adjustment at the edge server. To magnify the overhead, we change the frame rate from 60fps to 30fps and back to 60fps every 6 frames, which is much shorter than the usual adjustment interval. We then measure the CPU and memory utilization of the cloud gaming application and encoder by sampling the CPU processing time and application private bytes with the `typeperf` [222] every 1 second. We measure for 30 minutes to eliminate the randomness. We compare the scenario with a stable frame-rate of 60fps (`stable`) and a frequently switching frame-rate (`switch`) in Figure 6.18. For CPU utilization, both scenarios have a similar distribution from 0% to 20%. `switch` is a little better than `stable` since producing a lower frame rate takes fewer CPU resources for the gaming application. As for memory utilization, the major memory consumption is from the gaming application. Frame-rate switching slightly increases the utilization of private bytes since frequently resetting the encoder requires allocation of memory. Nonetheless, the increase of memory utilization is less than 1.8% even at the 99%ile, which is negligible and could be even lower in the case of normal frame-rate adjustments.

Image quality degradation. We also investigate the potential image quality degradation caused by AFR. We record two raw videos from games, one in a running scene (R) and another in a standing scene (S). For each video, we switch the frame rate every 100 frames 15 times and measure the video quality for the following 400 frames. We investigate three video quality metrics, peak-signal-to-noise-ratio (PSNR) [14], structural similarity index (SSIM) [258], and video multimethod assessment fusion (VMAF) [166], and present the results in Figure 6.19. `stable` and `switch` denote the scenarios where the frame-rate remains unchanged or frequently switched. Results demonstrate that frequently switching the frame rate will not affect the video quality: the video quality of two videos on three metrics are comparable in all cases.

6.6.5 Deployment in the Wild

Finally, we evaluate the performance of AFR by deploying it onto Windows clients of our cloud gaming service, Tencent Start, in one of its production clusters. Before the deployment of AFR, our cloud gaming service follows the frame control strategy in WebRTC (i.e., `DropTail`). To make a clean and controlled comparison, we only present the results from online A/B tests in our production clusters, when all other implementations and settings are kept the same. The A/B test is conducted from January 8, 2021, to January 14, 2021, resulting in 5369 Ethernet sessions and 1467 WiFi sessions. The parameter settings of AFR remain the same as the simulation (§6.5.2). We randomly enable (or disable) AFR with a probability of 50% for each session, and present the results in Table 6.2. Similar to the simulation results, the ratio of stuttered frames measured by total delay (P(T>100ms)) in both categories has been improved by 34% and 30%, which significantly improves users'

Cat. (1)	Q99	Q>50ms	T99	T>100ms	Session
DropTail	54ms	1.11%	101ms	1.03%	7.30%
AFR	22ms	0.51%	80ms	0.68%	5.82%
Cat. (2)	Q99	Q>50ms	T99	T>100ms	Session
DropTail	64ms	1.83%	174ms	3.00%	24.00%
AFR	37ms	0.54%	160ms	2.11%	21.17%

Table 6.2: Performance of deployment in the wild. Metrics are the 99%ile of queuing delay (Q99), the ratio of frames with Q>50ms, the 99%ile of total delay (T99), and the ratio of the stuttered frame (T>100ms). `Session` is the ratio of sessions with stutter ratio >5%. Cat. (1) and (2) are Ethernet and WiFi on Windows clients.

experiences in interactive streaming. The stuttered sessions (with the same metric as Figure 6.14(a)) have also been reduced by 17% on average, indicating these users could be alleviated from stuttering streaming experiences. Therefore, the online deployment also demonstrates significant benefits of AFR for high-quality RTC users. AFR has already been deployed onto all production clusters of Tencent Start for over one year, serving thousands of users each day.

6.7 Discussions

In this section, we discuss the potential limitations of AFR.

Application scenarios. In this paper, we mainly evaluate the performance of AFR on traces or production clusters of our cloud gaming service. However, as we introduce in §6.1 and §6.2, the overload of decoder queue generally exists in many high-quality RTC scenarios, such as VR streaming or 4K live streaming, as long as they stream high frame-rate and high bit-rate video onto commercial clients. We evaluate AFR with cloud gaming due to

access to the real-world traces and production services X. We leave the deployment of AFR over other scenarios as our future work.

Coexistence of multiple control loops. There are other control loops that work simultaneously in the RTC system. For example, the underlying congestion controller will also control the bit-rate of the video based on network conditions [77]. The video codec will also adjust the quantization parameter based on the scenes to encode [49]. As we discussed in §6.3.2, these parameters are affected by different causes (network congestion, decoder degradation, scene variation), which are orthogonal to each other. Therefore, the adaption of the frame rate is orthogonal to the other controllers in the RTC system. In §6.6.5, we evaluate the performance of AFR with all these controllers in our real production in the wild. We leave the coordination of different controllers on the joint optimization over the user's experience for the future.

6.8 Summary

In this paper, we propose AFR to reduce the queuing delay of the decoder queue for high-quality RTC by dynamically adjusting the frame rate. AFR introduces a stationary controller and a transient controller to respectively mitigate the stationary heavy traffic and contingent arrivals and services. We further evaluate the performance of AFR with trace-driven simulations and deployments in the production clusters. Experiments demonstrate that AFR could significantly reduce the stuttering ratio and tail total delay.

7

Transport Layer on Data Path: Discriminating Retransmissions

7.1 Introduction

A major challenge to control the deadline misses comes from the high instantaneous loss rate on the Internet. Due to the spatial dependency within video frames and temporal de-

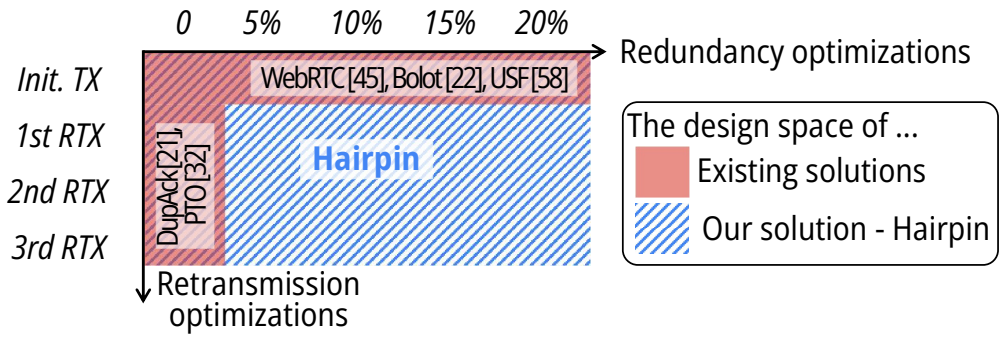


Figure 7.1: An illustration of the design space of existing solutions and Hairpin. By co-designing the redundancy and retransmission at the transport layer, Hairpin is able to break the existing trade-off between bandwidth cost and deadline miss rate.

pendency between video frames, interactive streaming expects packets to be reliably delivered [190]. However, from our measurement of our edge-based cloud gaming service in production with O(10,000) users, sessions can experience a drastically high *instantaneous* loss rate. Although the average loss rate is considerably low by mechanisms such as proper rate control, our measurement observes that more than 2% of video frames suffer from an instantaneous loss rate of 20% or higher (§7.2.1). It indicates that those lost packets are concentrated on a few frames. Thus, although the network RTT can be very low with edge deployments, retransmissions of lost packets take additional time and will consequently violate the deadline. Thus, it is essential to optimize the loss recovery mechanisms to control the deadline miss rate (DMR) of video frames.

Unfortunately, existing solutions to recover packet losses cannot meet the stringent DMR requirements with a reasonable bandwidth cost. As shown in Figure 7.1, one line of research efforts (the vertical dimension) is devoted to quickly retransmitting lost packets, such as probe timeout (PTO) [86], from the transport layer. However, merely retransmitting lost packets cannot meet the requirement of interactive streaming – the DMR is much

higher than 0.1% (§7.4.3). Another line of effort (the horizontal dimension) is devoted to adaptive forward error correction (FEC) so that the client might be able to recover packets based on redundant packets without retransmission [17]. Yet, redundancy-based solutions come with the price of a considerable bandwidth cost of 20% or more due to the high instantaneous loss rate. For content providers, such a high bandwidth cost will drastically increase operating expenses and degrade users' video quality. To the best of our knowledge, none of the existing solutions *jointly optimized* retransmission and redundancy. Such an orthogonal design of redundancy and retransmission, even when adopted together, still cannot meet the needs of bandwidth cost and DMR for interactive streaming.

Our key insight is to break the trade-off by discriminating retransmission packets. Edge-based interactive streaming services can achieve an average RTT of 10-20ms between application servers and users by deploying the servers on the edge [78, 197, 282]. In this case, limited times of retransmissions (but not too many) are tolerable for applications that have a deadline of 50-200ms (§7.2.1). But the strategy for retransmission packets must be different for the initial transmission packets. The volume of retransmission packets is much less than initial transmission packets since packet loss is always the minority. Yet, retransmission packets have a much tighter time requirement since they have already consumed time. This brings new changes to reduce the bandwidth cost and the DMR at the same time (§7.2.4). By discriminating the strategies for initial transmission and retransmission packets, we can break the trade-off between bandwidth cost and DMR.

Discriminating retransmissions for a different redundancy rate is the main insight for this paper, which will help a lot on the performance (§7.4.5). We then propose Hairpin*, a

*In badminton, a hairpin shot is played when the shuttle is very near to the ground and the net (the deadline of a shot) [243].

new packet loss recovery mechanism to jointly optimize packet retransmission and redundancy for edge-based interactive streaming (§7.3.3). However, as later elaborated in §7.3.2, to further analytically optimize the performance, we still face the challenge of (1) the dependency of decisions and future states, (2) the multi-dimensionality of decisions, and (3) the convoluted goal of DMR and bandwidth cost. In response, Hairpin further formulates the problem into a Markov decision process (MDP), which is known for efficiently optimizing the temporal dependency [253]. We then encodes the decisions and states into nodes of MDP to reduce the complexity and achieve the optimal result.

We conduct a week-long packet-level measurement campaign on Tencent edge-based cloud gaming service to motivate the design of Hairpin (§7.2.3 and §7.2.4). We then implement Hairpin and evaluate it with both trace-driven simulators and real-world deployments in production (§7.4.1). Experiments demonstrate that Hairpin could significantly push forward the Pareto frontier [2] by reducing the DMR by 67%-80% and achieve comparable bandwidth costs simultaneously compared with state-of-the-art baselines (§7.4.3). Preliminarily deploying Hairpin in Tencent cloud gaming service in production also shows significant and consistent performance improvements in different types of networks (§7.4.6). We will release the code and the traces of Hairpin.

Our main contributions are summarized as follows:

- We motivate the need for joint optimization of retransmission and redundancy through the operating experiences of a production edge-based interactive streaming service (§7.2).

- We present challenges in the joint optimization over retransmissions and redundancy for edge-based interactive streaming, and then propose Hairpin with MDP formulation (§7.3).
- We implement and integrate Hairpin in a cloud gaming application in production, and extensively evaluate its performance with trace-driven simulation and real-world deployments (§7.4).

7.2 Background and Motivations

We introduce the interactive streaming (§7.2.1), present our measurement of packet losses (§7.2.2), analyze why existing solutions are insufficient (§7.2.3), and motivate the design of Hairpin (§7.2.4).

7.2.1 Interactive Video Streaming

Interactive streaming applications are increasingly attracting interest in many scenarios. Examples include cloud gaming [12, 15, 19], remote driving [10, 170], cloud phone / PC [43, 57, 114], and regional videoconferencing [13], forming a considerable market value of billions of dollars. Compared with legacy live video streaming, with the intensive deployment on edge nodes (or content generators in VR), the network delay over the wide-area network could be reduced for interactive streaming (e.g., an average RTT of 10-20ms [78, 197, 282]). With the recent emergence of the metaverse and so on, these interactive video streaming applications are going to be increasingly dominant on the Internet. Edge-based interactive streaming imposes specific requirements on transport, as summarized below.

Stringent deadline requirements. Since interactive streaming applications continuously interact with humans, controlling end-to-end delay is critical for a seamless user experience. For example, videoconferencing may expect an end-to-end delay of $<$130ms for network [150, 188], while cloud gaming would argue for a latency of $<$96ms [151][†]. In practice, server- and client-side processing usually take $\approx$30 ms [45, 123, 239, 259]. Therefore, the end-to-end round-trip delay *for the network* should not exceed 50-150ms (depending on scenarios), which is the deadline required by the application [27, 241].

This also corroborates our measurement study with users in our production cloud gaming service. We measure our cloud gaming service in production for one week (details in Appendix D.1), with O(10,000) users every day, and collect a variety of metrics. Unless other specified, the analysis using online data in this paper is also from this measurement campaign. We categorize the measured round-trip interaction delay of each video frame into several intervals. We present the appearance distribution of the position of those frames in a flow for each category in Fig. 1.2, where the x-axis is the position of that frame in a session normalized by the length of that session. Compared to the uniform distribution of low-delay frames (solid lines), frames with an end-to-end delay of $>$100ms (dashed lines) have a higher probability to appear around the end of a flow. We hypothesize that this is because users tend to exit a session if they have a high end-to-end delay. User's exiting behavior is a critical metric for user's experience in real-time video streaming [85]. In the meantime, setting a deadline for the delivery and reducing the fraction of higher than that specific value has also been widely adopted in real-time video streaming [188, 190].

[†]Based on the statistics of the majority of people. Different users and applications could have different latency sensitivity. For example, for gaming applications, 3D games have more stringent latency requirements than 2D games [143]

The similarity between the 50ms and 100ms curve in Fig. 1.2 also indicates that, as long as packets could be delivered within the deadline (∼100ms in this case), faster delivery barely improves the user's experience.

Thus, we should minimize the *deadline miss rate* (DMR) to enable a seamless experience for users in interactive streaming, where in our cloud gaming service, the deadline for interaction delay is around 100ms. For interactive streaming, it is essential to minimize the occurrence of deadline misses for frames to an ultra-low level. For example, even a DMR of 10^{-3} still leads to a poor experience every 1000 frames (17 seconds at 60 fps), which drastically degrades the user's experience [27].

Reliable delivery. Meanwhile, interactive streaming also requires reliable delivery for each frame. For commercial video codec, failing to deliver a part of the frame will lead to severe image quality degradation. Moreover, the loss of one frame would also lead to blurring for the subsequent frames due to the dependency between frames[‡]. Therefore, existing interactive streaming services usually try their best to reliably deliver frames. For example, industrial frameworks (e.g., WebRTC) [17, 137] and academic efforts [66, 116, 208] propose to employ forward error correction (FEC) to recover lost packets at the receiver if possible, and will retransmit lost packets if the recovery fails [18].

Low bandwidth cost. The bandwidth cost is still one of the largest operating expenses in our and other cloud gaming service [55]. Moreover, to achieve a satisfactory user experience, interactive streaming must stream with high video resolution and frame rate (e.g., 60fps and >1080p for cloud gaming), which requires high goodput to support. Given the

[‡]Mechanisms such as scalable video coding (SVC) allow limited packet losses, yet reduce the bandwidth efficiency and require client support [234].

requirements of low operating expenses and high video quality for users, we need to control the bandwidth cost in packet loss recovery.

7.2.2 Packet Losses in Edge-based Interactive Streaming

Our observation from our cloud gaming service is that although the median loss rate is as low as 10^{-3}, the *instantaneous* loss rate could be very high. In our measurement campaign as described in §7.2.1, we also calculate the *session-level loss rate*, which is the ratio of total lost packets in one user session (minutes to hours, containing at least O(10,000) frames), to reflect the average loss rate over a long timescale. We then calculate *frame-level loss rate*, which is the ratio of lost packets within one frame (tens of milliseconds), to show the instantaneous loss rate over a short timescale. For example, if a session has 1M packets and 10 of them are lost, the session-level average loss rate is 0.01%. Meanwhile, if these 10 packets belong to the same video frame which has 50 packets in total, the frame-level instantaneous loss rate will be 20% for that frame and 0% for other frames.

As shown in Fig. 1.3, the session-level loss rate is 0.05% at the median, which is comparable to similar measurements [132]. However, the instantaneous frame-level loss rate could be very high: 2% frames lose more than 20% of their packets within one frame. Such a high instantaneous packet loss poses a great challenge in controlling the deadline miss rate to 10^{-3} or lower – we can no longer ignore these transient behaviors and have to deliver video frames in time even when the instantaneous loss rate is high.

Moreover, these packet losses cannot be easily mitigated by reducing the sending rate. To achieve a low latency, most CCAs in interactive streaming use *delay* as the signal to reduce the sending rate (e.g., BBR [75], Copa [47], GCC [77]). In this case, congestion losses

rarely happen since the sending rate has already been reduced based on an increasing delay in advance, which has also been measured in related work [77]. Our online measurements unveil similar observations: our cloud gaming service has already adopted a delay-based CCA similar to GCC [77], which is widely deployed in interactive streaming applications such as Chrome and Stadia. We further demonstrate the weak correlation between RTT[§] increases and packet losses in our measurement in §7.2.4. As shown in Fig. 1.3, losses are still outstanding at the tail, indicating that merely controlling the bit rate or frame rate is still insufficient to avoid packet losses for edge-based interactive streaming.

7.2.3 Why Existing Solutions Fail?

As we discussed in §7.1, packet losses contribute a lot to deadline misses. Thus, we investigate why existing packet loss recovery mechanisms are insufficient for edge-based interactive streaming. Existing solutions mainly fall into two categories as follows.

Retransmissions. Existing transport protocols (e.g., TCP) rely on retransmissions to cope with packet losses. Merely relying on retransmissions is insufficient to achieve an extremely low DMR for interactive streaming frames at the magnitude of 0.1% or lower. For example, when the packet loss rate is instantaneously 20%, there would still be 0.16% packets lost even after 3 retransmissions. Note that since there could be tens to hundreds of packets per frame, being unable to deliver even one packet would violate the deadline requirement of that frame since interactive streaming requires all packets to be reliably delivered (§7.2.1).

[§]In this paper, we use RTT to represent the delay at the network layer that does not contain the time of retransmission. We use application delay to refer to the delay at the application layer that contains the retransmissions.

Thus, the DMR of frames is still considerably high when relying on retransmissions and rate controls only. Our evaluation in §7.4 also demonstrate the performance degradation.

Redundancy-based algorithms. There are also several solutions in interactive streaming with redundancy mechanisms such as FEC. However, existing adaptive FEC solutions from both the industry [17, 137] and academia [66, 116, 208] optimize the FEC parameters only for the initial transmission. They adjust the number of FEC packets according to loss rate and retransmit packets as usual when packet loss occurs. Note that packet losses are not deterministic: when the transient loss probability increases to 20%, it does not mean precisely one packet loss every five packets. In this case, to achieve an extremely low DMR of 10^{-3} or lower, FEC rates need to be much higher than the loss rate, leading to severe bandwidth cost (§7.4). For example, WebRTC, a state-of-the-art interactive streaming framework, will send 100% redundant packets during this short timescale of high instantaneous loss rate for initial transmissions. In this case, there will be considerable bandwidth cost while the DMR might still not be satisfied. We further evaluate the performance of other baselines in §7.4.2.

7.2.4 Motivations

Therefore, with the reduced RTT, retransmissions are tolerable to some extent for edge-based interactive streaming. In this case, we have the following observations on how and what to retransmit.

RTT being much lower than the deadline enables the joint optimization of redundancy and retransmission. As we discussed before, with an RTT of 10-20 ms and a dead-

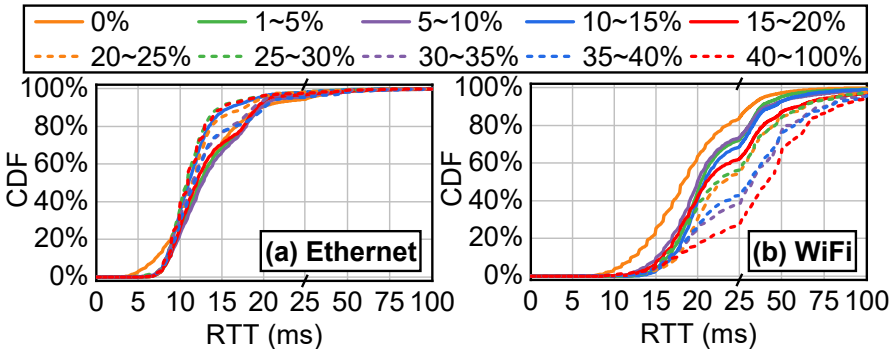


Figure 7.2: RTT distributions measured in production, categorized by the frame-level loss rate. Note that retransmissions are not counted.

line of 50-150 ms, multiple retransmissions are tolerable to some extent. This enables the joint optimization of redundancy and retransmission, which results in benefits in two folds:

- Reduce the deadline miss rate. In existing FEC mechanisms, many of the deadline misses come from the packet losses in the retransmissions. When adding redundancy packets over retransmission packets, we could effectively avoid the loss of retransmission packets and further reduce the deadline miss rate.

- Save bandwidth costs. To achieve the same DMR, the bandwidth cost of adding redundancy to retransmissions is significantly lower than that of only adding redundancy to initial transmissions. This is because retransmission packets are always the minority in bandwidth consumption – redundifying retransmissions will only introduce a little bandwidth cost, but could have significant DMR improvements.

When more rounds of retransmissions are tolerated (e.g., with smaller RTTs), the joint optimization will have more significant benefits (later presented in §7.4.4). We are thus

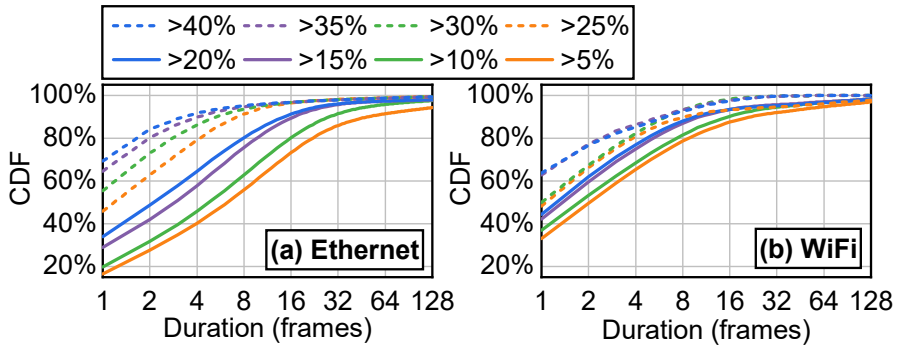


Figure 7.3: The distribution of the duration of each loss event measured in production. We measure the duration of each time when the loss rate is larger than different thresholds (5%, ..., 40%). Loss rates are measured at the frame level. The network type is reported from our cloud gaming clients. Better viewed in color.

motivated to utilize the retransmission chances enabled by edge deployments and jointly optimize the redundancy and retransmission mechanisms.

Loss recovery adaptions at the server are possible. Dynamically optimizing the tail cases of high instantaneous loss rate needs quick adaption. According to our measurement, the feedback loop between the server and client is smaller than the duration of loss events, making the joint optimization of redundancy and retransmission practical. This comes in two folds:

- The feedback loop does not inflate with the increase in the loss rate. We measure the RTT of our cloud gaming service and categorize them into different frame loss rate intervals. As shown in Fig. 7.2(a), the distribution of RTT does not significantly vary with the frame loss rate. The RTT in WiFi increases with the increase of frame loss rate (e.g., due to retransmissions at the link layer [84]). Nevertheless, even when the frame-level loss rate is 30% (the dashed green curve in Fig. 7.2(b)), 60% of those acknowledged packets have an RTT of less than 25ms. This indicates (i) the server is

able to quickly detect the network condition changes, and (ii) there are still multiple transmission chances when the instantaneous loss rate increases.

- The duration of loss events is transient but still longer than several feedback loops. We measure the duration of lossy frames in our cloud gaming service and present the results in Fig. 7.3. According to our measurements, most loss events span multiple RTTs. For example, 70% of frames with a frame-level loss rate of $>10\%$ will last more than 2 frames in Ethernet sessions, which is several times the median RTT (12ms) at the frame rate of 60fps. Therefore, the reaction from the server is still effective to alleviate packet losses by adjusting the redundancy parameters.

7.3 Hairpin Optimizer

As we discussed above, edge-based interactive streaming needs to reduce the deadline miss rate and bandwidth cost. For clarity, we first present the formula of frame deadline miss rate (DMR) and bandwidth cost (BWC):

$$\begin{aligned} DMR &= \frac{\#\text{Frames arrive after the deadline}}{\#\text{Total frames}} \\ BWC &= \frac{Redundancy_{byte} + Retransmission_{byte}}{Data_{byte}} \end{aligned} \tag{7.1}$$

A higher DMR or BWC means more frequent stutters or higher operating expenses respectively, both of which interactive streaming service providers will try to avoid. Note that pushing DMR to an extremely low level is critical since the lower it is, the better user's experience is going to be.

In this section, we first summarize some intuitions in the design space of joint optimization of redundancy rate and retransmission and present a strawman solution (§7.3.1). We then present the design challenges in the joint optimization of retransmission and redundancy (§7.3.2). We address these challenges by providing a Markov chain-based optimization algorithm to efficiently improve both the DMR and BWC (§7.3.3). We finally discuss how Hairpin handles the inaccuracy in measurement, the overhead in online deployment, and other practical issues in §7.3.4.

7.3.1 Basic Idea and Strawman Solution

Discriminating retransmissions from initial transmissions. The most important insight in this paper is to understand the significance of discriminating retransmissions from initial transmissions. In other words, we want an adaptive redundancy rate based on the planning of multiple transmission chances. The short RTT of edge-based interactive streaming enables packets to have more than one transmission chance without violating the deadline. The ratio of RTT and remaining time t indicates the potential number of (re)transmissions. For example, when the current RTT is 20ms and packets still have 40ms towards their deadline, the ratio follows $\frac{t}{RTT} = \frac{40ms}{20ms} = 2$, indicating that these packets could be approximately transmitted twice before the deadline. Packets with more transmission chances could better utilize the potential retransmissions to deliver packets before the deadline, which has already been discussed in §7.2.4. Therefore, our basic idea is to take future transmission chances into consideration when optimizing the redundancy rate. When one batch of packets has more foreseeable transmission chances (i.e., the deadline is still far away), we could reduce the redundancy rate to save bandwidth costs. When the remaining time

of these packets is getting closer to the deadline due to retransmissions, we could further increase the redundancy rate to avoid deadline misses.

Strawman solution: RTT-aware adaptive FEC algorithm. Therefore, a strawman solution is to (i) add redundancy to both initial transmissions and retransmissions, and (ii) consider the remaining transmission chance in the optimization of the redundancy rate. Since there have already been existing solutions on the redundancy rate based on network conditions [17, 66, 208], we could introduce a multiplier controlled by the transmission chance over the existing redundancy rate optimizations, i.e. a strawman solution is to reduce the redundancy rate when there are many transmission chances, and increase it when transmission chances are few. Thus, we could enhance these algorithms by introducing a factor over the results from existing algorithms.

FEC consists of two parameters (d, k), where d data packets and k redundant packets are sent as a *block*. Block is composed to the convenience of FEC encoding. If there are up to k packets lost in an FEC block (d, k), an ideal FEC decoder can recover all data packets with any remaining packets [220, 221, 274]. We denote $\beta = \frac{k}{d}$ as the FEC redundancy rate, and d as the FEC block size.

Specifically, given a packet loss rate α and bitrate B, assume one of the state-of-the-art solutions has already determined that $\beta_0(\alpha, B)$ should be the optimized redundancy. We could then increase or decrease the redundancy rate $\beta_0(\alpha, B)$ based on the remaining transmission chance $\frac{t}{RTT}$, i.e.:

$$\beta(\alpha, B, RTT, t) = k \cdot \frac{RTT}{t} \cdot \beta_0(\alpha, B) \tag{7.2}$$

where k is a coefficient to adjust how aggressive the strawman solution is going to increase or decrease the redundancy rate.

In fact, according to our evaluation in §7.4.5, such a strawman solution is enough to push the Pareto frontier of DMR-BWC forward. However, it confronts a series of shortcomings, which prevents the operator from further improvements in performance. We will elaborate on these challenges in the following section.

7.3.2 Design Challenges

Although we have presented a heuristic RTT-aware adaptive FEC algorithm as above, it is still challenging to optimize these parameters due to the following reasons.

Temporal dependency: cascading decision-making between transmission rounds. When considering multiple transmission chances, the decision of FEC parameters of one round of transmission would cascadingly affect the optimization of the next round. For example, if we aggressively add a high redundancy rate to a group of packets, the number of packet losses will then be decreased. On the contrary, a low redundancy rate for the same group of packets would probabilistically increase the number of packet losses under the same network condition. However, these packet losses bring more packets to retransmit in the next round. If we consider all actions for F packets for the foreseeable L rounds of transmission, the action space will be extremely large: Since for each redundancy decision, there are F possible scenarios of the number of packets to transmit in the next round (depending on how many packets are lost), the number of variables that we need to optimize

will be $O(F^L)$[¶]. Therefore, in the enlarged action space over multiple retransmissions, it is challenging to efficiently optimize. Moreover, the conditional probability between scenarios is not linear (e.g., hypergeometric for individually independently and identically distributed losses). Therefore, using traditional optimization methods such as integer programming in an extremely large action space is impractical. We need to coordinate the choices in different rounds of transmission to achieve optimal performance.

Spatial dependency: redundancy rate and block size are tightly coupled. Even in a single round, different variables (e.g., redundancy rate, block size, etc.) still have complicated dependencies on each other. This goes to the following aspects:

(a) Number of packets to transmit in one round affects redundancy rates. The number of packets to transmit in the different rounds is varying, depending on how many data packets are lost during the last transmission. The penalty of redundancy rate on BWC also varies according to the number of packets to retransmit. For example, when there are few packets to retransmit, even adding a redundancy rate of 100% for retransmissions would not consume too much bandwidth, as also discussed in §7.2.4. Therefore, fewer data packets to retransmit would encourage a more aggressive redundancy rate. The strawman solution is not aware of the dependency here, leading to its suboptimal result.

(b) Dispersion of blocks might lead to deadline misses when using larger blocks. Due to the bandwidth limit at the bottleneck link, packets sent out at the same time could be dispersed [146] and arrive at the receiver one by one. In this case, constructing large blocks

[¶] For a frame with 50 packets (F=50), and 5 potential transmission rounds (L=5, e.g., RTT is 20ms and deadline is 100ms), this turns into 10^8 variables.

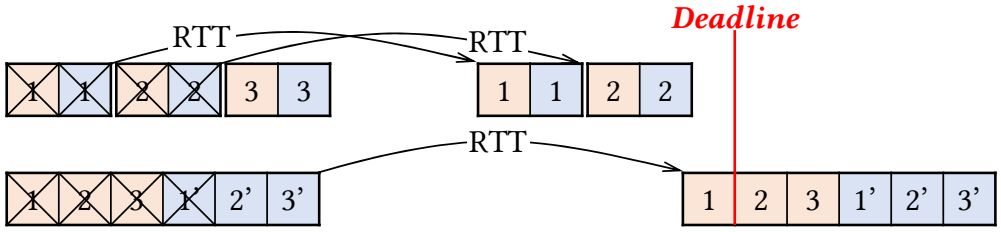
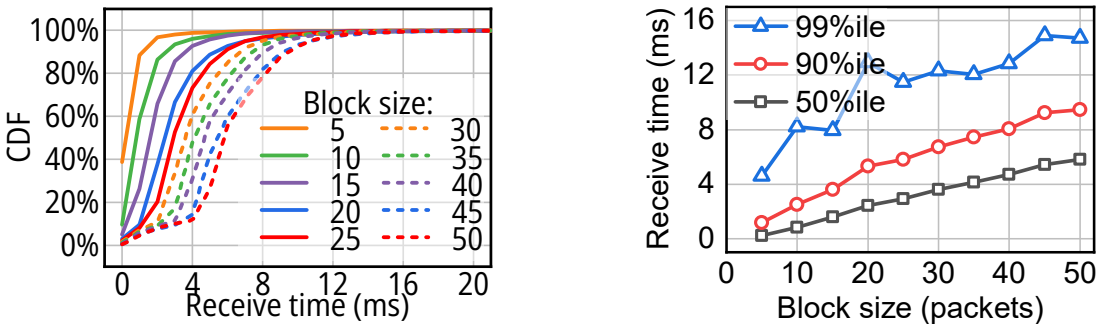


Figure 7.4: Smaller block sizes in one frame could have better performance. Scenarios above and below represent using small and large blocks. Data and FEC packets are shaded orange and blue.



(a) Cumulative distribution.

(b) Trend of percentiles.

Figure 7.5: Block receiving time with different block sizes. FEC blocks are burstily sent out at the server side. Fig. 7.5(b) is processed from Fig. 7.5(a). Measurement details in §7.4.2. Better viewed in color.

will increase the delay to wait for all packets at the receiver. Since packet losses can only be determined after the completion of one block, smaller blocks may know earlier whether they need retransmission and enjoy additional transmission chances before the deadline. For example, in Fig. 7.4, due to the early determination of packet loss, the retransmission of data packets for small blocks could arrive at the receiver before the deadline, while no packets could arrive before the deadline for large blocks. We quantify the influence of block size by measuring the receiving time of FEC blocks from our service online with different block sizes. As we can see in Fig. 7.5 and 7.5(b), with a block size of 50 packets, more than 10% blocks could span 10ms at the receiver, which is even comparable to the RTT. Also, smaller block sizes might also be beneficial when the loss rate is higher than the redundancy rate. As illustrated in Fig. 7.4, when the first four packets are lost during the transmission, data packet #3 could still be successfully delivered for a small block size (the case above in Fig. 7.4). For large blocks, there is no way to recover any lost packet if the loss rate is larger than the redundancy rate.

Convoluted goal: deadline miss rate and bandwidth cost. Unlike latency or throughput which we can directly measure, the estimation of the expected deadline miss rate needs to consider multiple potential rounds of transmission. In this way, the strawman solution, without explicitly estimating whether that frame is going to miss the deadline or not, will have suboptimal results. For example, the relationship between the packet loss rate and the success rate of delivering a video frame with tens of packets in a single round is *hypergeometric*, even under the identical and independent distribution (i.i.d.) assumption. Considering multiple future rounds together will only make the relationship between deadline miss rate and network conditions more convoluted. Moreover, some applications or

even the same application in different operating regions may have different preferences over deadline miss rate v.s. bandwidth cost. The traffic cost in some regions might be higher than in another, and some applications may give it all for the user's experience while others may not. Therefore, we need to explicitly optimize towards the goal to achieve the optimal result.

7.3.3 Model Formulation and Optimization

We have the following designs to address the challenges above.

Encode the temporal dependency in multi-round planning into edges in Markov chain. Markov chain is widely used in the optimization of the sequential decision-making process (e.g., reinforcement learning [253]). With the Markov chain, we can formulate the loss detection between two rounds of (re)transmission into the transition between two Markov nodes. In this case, by only focusing on the optimal parameters between the transition of the current state and its potential states in the next round, we could decouple the cascading effects of the transitions between neighbor nodes, which reduces the action space significantly. We further show in Appendix D.2.1 that, in such a Markov chain, locally focusing on the neighbor nodes could still have globally optimal results.

Encoding the spatial dependency between variables into nodes in Markov chain. To ensure the number of packets to transmit is considered in the optimization, we build a 2-D Markov chain, with two dimensions as the transmission chance and the number of packets to transmit. We present the state transition of our Markov chain in Fig. 7.6. Each node is represented by (d, l), where d denotes the number of remaining data packets to transmit,

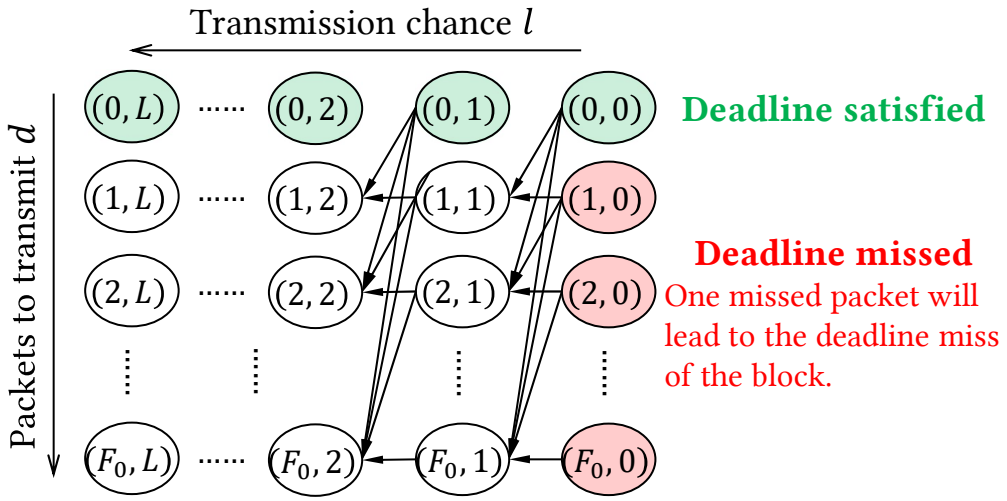


Figure 7.6: The absorbing Markov chain in redundancy rate optimization at given loss rate and frame size. l is the estimated remaining transmission chances for the packets to transmit.

and l represents the remaining transmission chance for those packets. Our goal is to find out the optimal redundancy rate for node (B, L), where B is a given block size, and L is the remaining transmission chance from Eq. 7.3. In this case, both the temporal dependency and spatial correlation between variables could be formulated into this 2-D Markov chain.

Explicitly optimize deadline miss rate and bandwidth cost with Markov chain formulation. We finally provide an explicit expression of the deadline miss rate and bandwidth cost for multi-round optimization within the formulation of MDP. We inversely calculate the DMR and BWC at different states from the last chance to transmit (as the last layer of the Markov chain), to the first chance to transmit (as the first layer of the Markov chain). In this way, the transition probabilities between states could be directly iterated. We further decouple the optimization of redundancy rate and block size to improve the optimization efficiency.

Notation	Explanation
Inputs:	
α	Network loss rate.
T	Remaining time till the deadline.
RTT	The network round-trip time.
Θ	The network bottleneck bandwidth.
F	The frame size of that frame.
Intermediate variables:	
L	Remaining transmission chance.
$l(n,r)$	The number of lost packets at the r-th layer with n data packets.
$k(n,r)$	The number of redundant packets at the r-th layer with n data packets.
DMR	Deadline miss rate.
BWC	Bandwidth cost.
Outputs:	
β_i	Redundancy rate at the i-th layer.
b_i	FEC block size at the i-th layer.

Table 7.1: Notations in §7.3.

We present the analytical model and the algorithm below. In interactive streaming, frames are continuously generated and sent out from the server. There are thousands to millions of frames within one stream, depending on the specific application, where the retransmission of previous frames overlaps with the transmission of subsequent frames. Therefore, similar to the finite element analysis in mechanics [21], we pick one frame from the stream, and analyze the expected DMR and BWC of that frame. The expected DMR and BWC of one frame should be consistent with the DMR and BWC of a stream. We list all notations that are going to use in Table 7.1. Specifically, Hairpin optimizes the FEC parameters as follows:

Step 1: Calculating remaining transmission chance. Given current network RTT, the remaining time towards deadline T, the bottleneck bandwidth Θ, and a certain block size d, the remaining transmission chance L could be calculated as:

$$L = \frac{T - d/\Theta}{RTT} \tag{7.3}$$

Step 2: Generating absorbing Markov chain. We then calculate the optimal redundancy rate given the current loss rate α and frame size F. We iteratively calculate the absorbing Markov chain from layer $l-1$ to layer l. We leave the detailed equations to Appendix D.2.1. For the node (d, l), at a certain redundancy rate β, its DMR follows:

$$DMR(d, l; \beta) = \sum_{d'=0}^{d} p((d, l) \to (d', l-1); \beta) \cdot DMR(d', l-1) \tag{7.4}$$

where $p((d, l) \to (d', l-1); \beta)$ is the transition probability from (d, l) to $(d', l-1)$ and could be calculated based on the current loss rate α and redundancy rate β (details in Appendix D.2.1). Similarly, the BWC could also be updated as:

$$BWC(d, l; \beta) = \beta \frac{d}{F} + \sum_{d'=0}^{d} p((d, l) \to (d', l-1); \beta) \cdot BWC(d', l-1) \tag{7.5}$$

where the latter term is the additional BWC introduced in this layer l. Then, we calculate the optimal β for (d, l):

$$\beta_{opt}(d, l) = \arg\min_{\beta} utility(DMR(d, l; \beta), BWC(d, l; \beta)) \tag{7.6}$$

and have $DMR(d,l) = DMR(d,l;\beta_{opt})$ and $BWC(d,l) = BWC(d,l;\beta_{opt})$. Here, $utility(DMR, BWC)$ is the utility function to balance preference for low DMR and low BWC. For simplicity, we adopt a linear combination of DMR and BWC as the optimization goal:

$$utility(DMR, BWC) = DMR + \lambda \cdot BWC \tag{7.7}$$

Note that Hairpin does not fall into the same trade-off between DMR and BWC as baselines, but improves both DMR and BWC, as we will evaluate later in §7.4.3. In practice, service providers can adjust the coefficient λ to balance stuttering events and bandwidth costs in different scenarios. A lower λ indicates that users prefer the deadline miss rate more than bandwidth costs. We also evaluate performance with different utility functions in §7.4.4.

Therefore, the redundancy rate for (B, L) could be optimized accordingly. After calculating all nodes at the layer l, we could then calculate the DMR and BWC at the layer $l+1$, until the node (B, L) has been calculated. Since the iterations between nodes are linear, as long as the utility function is monotonic to DMR and BWC (e.g., linear relationship), the optimality still holds.

We set $DMR(d, 0)$ to 1 for $d > 0$ since one missed packet would lead to the miss of the block (shaded green). We also set all $DMR(0, l)$ to 0 since there is no remaining packet to transmit. The BWC for all these boundary nodes is set to 0. Note that different block sizes and remaining transmission chance could multiplex the same chain to accelerate the optimization, since the chain only depends on loss rate α and frame size F.

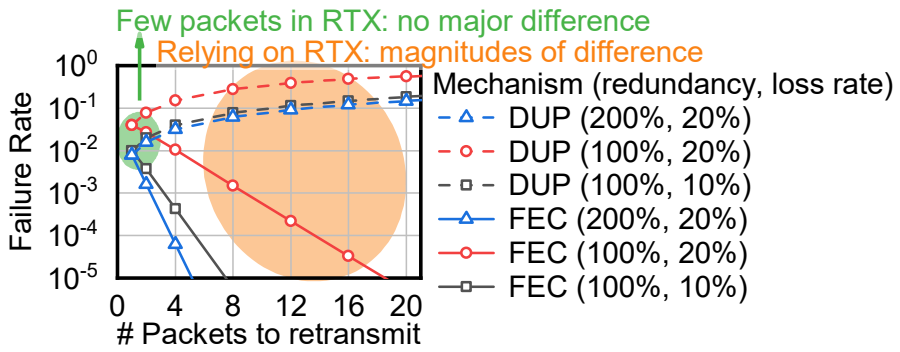


Figure 7.7: A theoretical illustration of the failure rate of retransmitting different numbers of packets by per-packet duplication or constructing FEC blocks. The failure rate of DUP increases with the number of packets to retransmit, since we need to ensure every data packet is delivered. We vary the redundancy rate and loss rate.

Step 3: Calculating optimal block size. We enumerate the possible block sizes from 1 to the frame size, calculate the DMR and BWC for each block according to the chain in Step 2, and finally find the optimal block size in terms of a given utility function. We leave the mathematical details to Appendix D.2.2. According to our evaluation in §7.4.5, not surprisingly, when the bottleneck bandwidth is high (i.e., the dispersion is insignificant), the optimal block size for most scenarios is the frame size. Nevertheless, when the dispersion is significant, constructing smaller blocks could achieve better DMR. Operators could optimize the block size for improvements at the last mile.

During the optimization of block sizes, we also optimize the trade-off of when a loss has been detected, whether to retransmit that packet as soon as possible or wait for other packets to formulate an FEC block. On recovery ability, constructing several lost packets into one FEC block might be more effective than individually retransmitting (or duplicating, if with redundancy) each packet. We calculate the failure rate of delivering these packets when there are different numbers of packets to retransmit at different redundancy rates

and loss rates and present the results in Fig. 7.7. When there are few packets that need retransmission, whether duplicating or constructing FEC blocks has no major difference (dashed line and solid lines shaded green). However, when optimizing at the *tail* for interactive streaming, there could be multiple packet losses within one frame. Therefore, considering each frame could contain tens of packets, it is possible to suffer losses of 4 packets or more at the tail. Constructing FEC blocks for these retransmission packets could reduce the failure rate of delivering packets by several magnitudes.

Step 4: Getting the optimal parameters. Finally, based on network conditions and remaining time towards a deadline, Hairpin can calculate the optimal block size based on Step 3, and the optimal redundancy rate with the block size based on Step 2.

7.3.4 Deployment Discussions

In §7.3, we analytically optimize the FEC parameters given certain network conditions. The reality might be more complicated than the theoretical model. In this section, we discuss several practical concerns of Hairpin based on our operational experiences. Our further trace-driven simulation and deployments in production in §7.4 also demonstrate the effectiveness of Hairpin in the wild.

Reducing computational overhead online. Hairpin adopts an optimization-based algorithm, which might not scale to production-scale deployments in terms of computational overhead. Since the optimization needs to run frequently (approximately every frame) and scale to tens of thousands of users simultaneously, it should be computation-efficient and time-efficient. In response, we do an offline step of enumerating the state space and solving

each specific instance. Then, in the online step, the algorithm will be reduced to a simple table lookup towards pre-computed optimized redundancy parameters. We enumerate the state space of Hairpin as below.

1. Remaining transmission chance: 1 to 10.
2. Loss rate: 0% to 50% with quantization of 1%.
3. Frame size: 5 to 60 packets with quantization of 5 packets.
4. Number of packets to (re)transmit: 5 to 60 packets with quantization of 5 packets.

Hairpin then stores the best redundancy rate and block size under different conditions. We found that the benefits of finer quantization are marginal. Our further evaluation in the real world in §7.4.6 shows that the memory consumption (2MB) and table lookup time are negligible for online deployment.

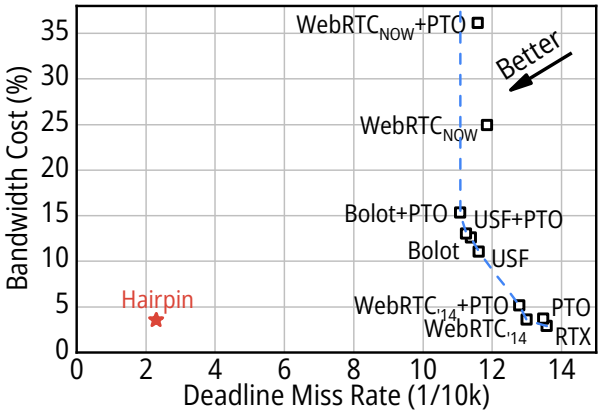
Handling network fluctuations. We discuss how Hairpin handles the fluctuations in network conditions. For RTT, as presented in Fig. 7.2, RTT does not increase too much – the median RTT always allows Hairpin to have 3-5 transmission chances no matter the loss rate. Moreover, we further measure the network conditions in Hairpin with a short sliding window to make sure Hairpin has the most recent network conditions. We set the measurement window to 2 frames and evaluate the sensitivity of this parameter in §7.4.4. In this case, the transient fluctuation of RTT could be reflected in the optimization results immediately. We later demonstrate in §7.4 that Hairpin behaves well with real-world traces and production deployments.

Handling various loss patterns. In this paper, when given a certain loss rate, Hairpin assumes the pattern of packet losses is identically and independently distributed (in the transition probability of Eq. 7.4). Note that the duration of a certain loss rate still follows the results of the online measurement in Fig. 7.3. In practical deployment, working with FEC codecs that could recover from different loss patterns (bursty or arbitrary) [220], Hairpin could also handle different loss patterns since Hairpin only focuses on how many packets within a block are lost. Since our data is collected frame by frame, if the burstiness spans over several frames, it will be directly reflected on the value of loss rates. If the burstiness spans within the frame, no matter how the pattern changes, the number of lost packets will not change, which does not affect the recovery efficiency of the FEC codec. For example, when there are 4 packet losses in one block, no matter whether these losses are consecutive or separated in the block, as long as there are 4 additional FEC packets in the same FEC block, the client would be able to recover these packet losses. Therefore, Hairpin does not rely on the assumption of underlying loss patterns, but only focuses on the number of lost packets. Packet losses might be consecutive across several frames. In this case, due to the short feedback loop enabled by edge deployments, Hairpin should have already timely reacted as analyzed in §7.2.4.

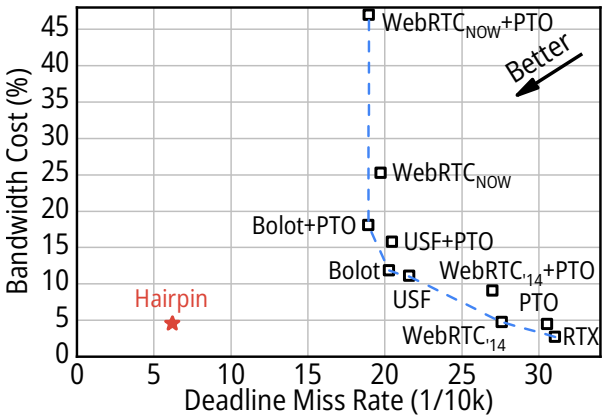
7.4 Evaluation

We introduce the implementations in §7.4.1 and experiment settings in §7.4.2. We further answer the following questions:

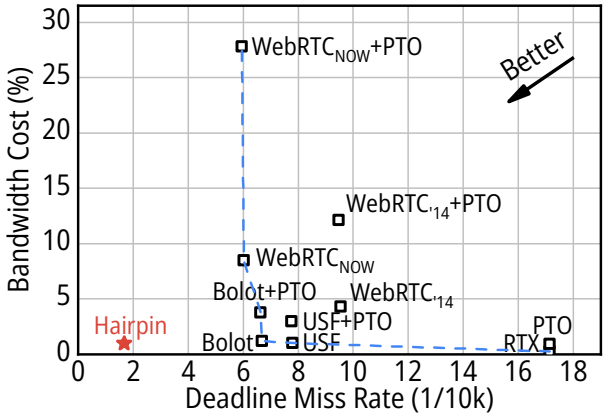
- How does Hairpin perform under real-world traces? We demonstrate that Hairpin could push forward the Pareto frontier of baselines on DMR and BWC (§7.4.3).



(a) Ethernet traces.



(b) WiFi traces.



(c) Cellular traces.

Figure 7.8: Trace-driven simulation. The blue dashed line is the envelope of all baselines on the Pareto frontier.

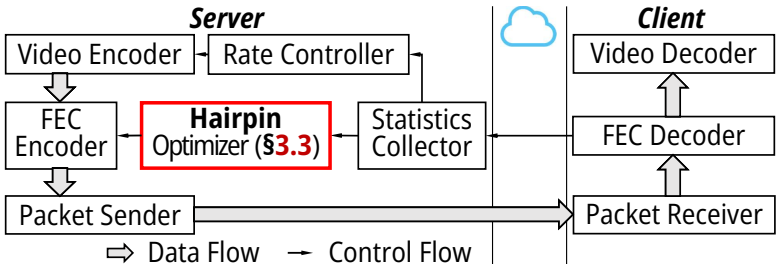


Figure 7.9: Overview of Hairpin implementation.

- Is Hairpin sensitive to the settings of parameters? We investigate the performance variation of Hairpin with different parameters, and demonstrate that Hairpin has performance improvements in a wide range (§7.4.4).
- Why could Hairpin outperform other baselines? In §7.4.5, we break down the performance improvements of Hairpin.
- How does Hairpin perform well in the wild? Finally, we deploy Hairpin in production servers and find that Hairpin significantly improves both DMR and BWC in the real world (§7.4.6).

7.4.1 Hairpin Implementation

We implement Hairpin in both an ns3-based WebRTC simulator [279] and our cloud gaming application in production. We present the workflow of Hairpin in the network stack in Fig. 7.9. Without Hairpin, interactive contents are first encoded with `Video Encoder` by the application, and then sent out at the transport layer frame-by-frame. Then the video frames could be received by the protocol stack at the client. `Packet Sender` and `Packet Receiver` abstract the network stack at the transport layer for connection management. After that, `Video Decoder` decodes the streaming contents and displays them to users. Meanwhile,

network conditions (e.g., RTT, packet loss events) will be measured at the server, collected by the `Statistics Collector`, and reported to `Rate Controller` to adaptively adjust the streaming bit-rate according to network conditions [77]. Hairpin inserts between the existing application layer and transport layer, and optimizes the redundancy parameters based on current network conditions, as shown in Fig. 7.9. The network statistics is still passed to the congestion controller (rate controller) without modification. The underlying transport protocol in our cloud gaming service is a customized version of the RTP protocol [233] based on UDP to allow the loss of redundant packets without modifying the kernel at the client. We implement Reed-Solomon FEC due to its recovery performance when the redundancy rate is $<100\%$ [220], and implement a customized FEC codec for the redundancy rate of $>100\%$. Note that Hairpin could also work with other codecs (e.g., XORFEC, FlexFEC, etc.) as long as their parameters are exposed to Hairpin. We discuss FEC codecs in Appendix D.3.

7.4.2 Experiment Setup

Traces. As for simulation traces, we collect a dataset in one production server in the wild on our cloud gaming service in two weeks in January and August 2021, resulting in more than 100M video frames and more than 600 hours of playtime. This also supports our measurements in §7.2 and §7.3. Users access the service via either Ethernet, WiFi, or cellular connection, which we collect from our cloud gaming client. The cloud gaming service streams at the frame rate of 60fps and the bit rate ranging from 2Mbps to 30Mbps. The network conditions are recorded on the server of our cloud gaming service, including the average RTT, average bit rate, and loss rate at the frame level (approximately every 16 ms).

The traces contain 1,995 Ethernet gaming sessions, 741 WiFi sessions, and 572 cellular sessions in total, each lasting from minutes to hours. To the best of our knowledge, we are the first to collect online traces from an interactive streaming service for weeks at both the frame level and packet level.

Baselines. We orthogonally review the public adaptive FEC mechanisms and retransmission mechanisms with deployments in practice. On the axis of retransmission optimization, we implement the following baselines.

- `Out-of-order`. Traditionally, packet losses are detected by checking the out-of-order packets, such as TCP duplicated ACK [64]. We use it as our default loss detection mechanism.
- `Probe timeout (PTO)`. Besides, to quickly detect packet losses of tail packets, recent researchers also propose an aggressive timeout-based loss detection mechanism [86].

On the axis of redundancy parameter optimization, we implement the following mechanisms:

- `WebRTC`$'_{14}$ comes from the research paper published by Google in 2014 [137].
- `WebRTC`$_{NOW}$ is the adaptive FEC mechanism used in WebRTC now (adopted by Google Stadia [96], Meet [67], etc.), replacing the `WebRTC`$'_{14}$. The difference is that `WebRTC`$'_{14}$ is aware of RTT and will reduce the redundancy rate when RTT is low, while `WebRTC`$_{NOW}$ is more aggressive on adding redundancy. We migrate the implementation of the m88 version of Chromium released in December 2020 [18].

- Bolot [66] and USF [208] are two heuristic adaptive FEC algorithms from the research community. Unlike Hairpin, they do not add redundant packets for retransmissions.

- RTX adds no redundancy, but fully relies on retransmissions.

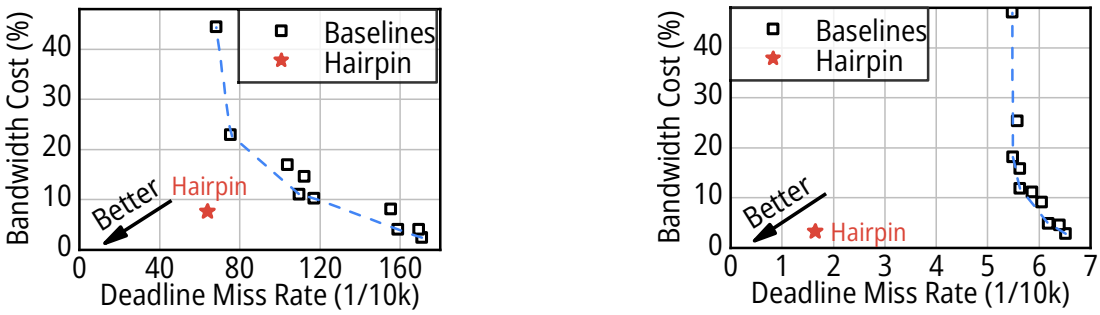
Note that none of these baselines optimize the redundancy for retransmissions here. Since these two lines of work are orthogonal to each other, we combinatorially implement 2 (retransmission) $\times$ 5 (redundancy) $=$ 10 baselines.

Hairpin Setup. In our simulation, we set the coefficient in the utility function in Eq. 7.7 to $\lambda = 10^{-4}$, the measurement window of network conditions to 2 frames, and the deadline to 100ms. We evaluate the sensitivity of these parameter settings in §7.4.4.

7.4.3 Trace-driven Simulations

To evaluate the performance of Hairpin in dynamic network conditions, we simulate Hairpin over real-world traces as introduced in §7.4.2. We emulate the collected traces of loss rate and RTT with ns-3, and evaluate whether Hairpin could capture the network dynamics of loss and RTT variations and effectively adapt in real traces. We first present the trade-off between DMR and BWC over three sets of traces in Fig. 7.8.

As shown in Figure 7.8, RTX has the lowest bandwidth cost since RTX only retransmits a packet after it is lost. However, it also has the highest deadline miss rate among all baselines. Meanwhile, WebRTC$_{NOW}$ working with PTO has the lowest DMR among all baselines but also the highest BWC. Other baselines stay on the Pareto frontier in the trade-off between DMR and BWC. In contrast, Hairpin could break the trade-off and achieve a much



(a) Deadline = 50ms.

(b) Deadline = 200ms.

Figure 7.10: The performance of Hairpin and all baselines (labels omitted for brevity) on WiFi traces when the deadline requirement from the application is different.

better DMR and BWC, as the red stars denoted in Figure 7.8: 67%-80% lower than the lowest DMR (WebRTC$_{NOW}$), and comparable BWC as RTX. Thus, as we analyzed above, Hairpin could effectively improve both DMR and BWC significantly compared with all other baselines.

Note that the traces here are collected from our production servers, including the network RTT and instantaneous loss rate, with a fined granularity of every 16ms. The results in WiFi traces are worse than in Ethernet traces since WiFi traces have higher loss rates and RTTs, as measured in §7.2.4. Results over cellular traces are surprisingly good. This is because, during our online measurements, we just started to provide cloud gaming service for cellular users and had admission control over network conditions during that time. In all, Hairpin could significantly push forward the Pareto frontier of existing baselines in all traces.

7.4.4 Parameter Sensitivity

We also evaluate how Hairpin performs with different parameters.

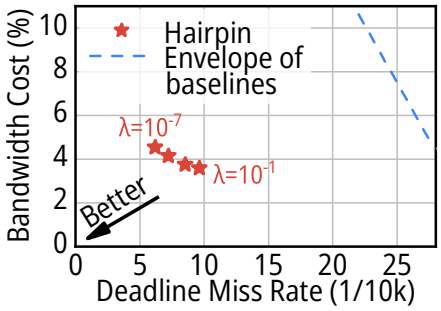


Figure 7.11: Parameter sensitivity of λ in the utility function of Hairpin.

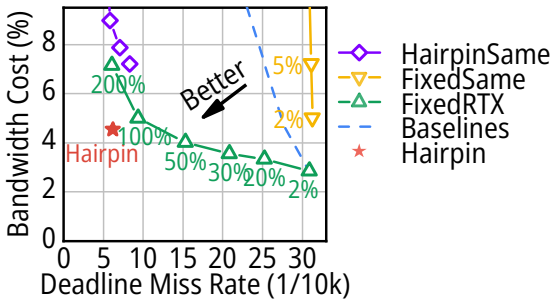


Figure 7.12: Discriminately handling retransmissions helps. The envelope is from Figure 7.8(b).

The setting of the deadline. In the evaluation in §7.4.3, the deadline is set to 100ms. We also investigate how Hairpin performs when the deadline is shorter or longer. Thus, we present the results of DMR and BWC of Hairpin and baselines over WiFi traces when the deadline is set to 50ms (Fig. 7.10(a)) or 200ms (Fig. 7.10(b)). As presented in Fig. 7.10, given the same trace, when the deadline is shorter (50ms), the advantages of Hairpin over baselines are a little less than when the deadline is 100ms. This is because the retransmission chance is less and the design space is smaller when the deadline is shorter. Nevertheless, Hairpin is still much better than all existing baselines. When the deadline is longer, the benefits are even larger due to the larger design space in retransmission. Results over other sets of traces are similar.

Utility coefficient λ. For the utility coefficient λ in Eq. 7.7, as introduced in §7.4.2, it could adjust the preference over the trade-off between the DMR and BWC. A higher λ indicates that users prefer the BWC more, while a lower λ indicates that the DMR is outweighing the BWC. Therefore, we change λ from 10^{-1} to 10^{-7}, and present the DMR and BWC of Hairpin with different λ over WiFi traces in Fig. 7.11. Note that Fig. 7.11 is

zoomed in from Fig. 7.8(b). As shown in Fig. 7.11, the BWC is decreasing with the increase of λ, while the DMR is increasing by a little. Thus, operators could adjust λ to balance the DMR and BWC according to the requirements of applications.

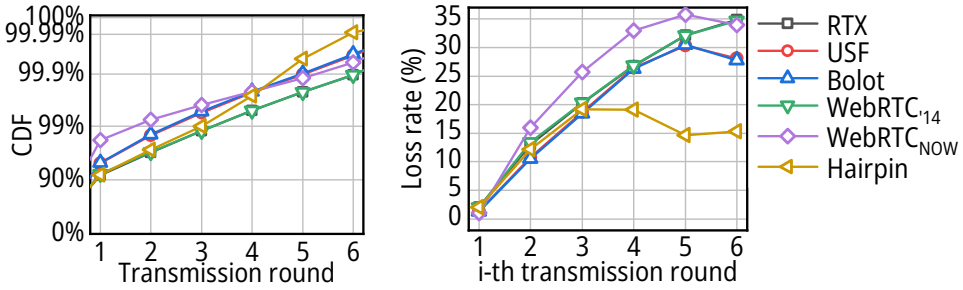
7.4.5 Hairpin Deep Dive

We further provide a deeper understanding of Hairpin in the following aspects.

The effectiveness of redundancy over retransmission. One of the major observations in this paper is to identify the significance of differently handling initial transmission packets and retransmission packets. To validate this, we further compare the performance with three baselines:

- **FixedSame** non-discriminately adds FEC packets to both initial transmission and retransmission packets with a specified fixed ratio.
- **FixedRTX** only adds FEC packets to retransmissions with a fixed ratio, and never adds FEC packets to initial transmissions, in contrast to all existing solutions in §7.4.2.
- **Hairpin-Same** uses exactly the same Markov-chain-based formulation as in Hairpin, but does not discriminate the initial transmissions and retransmissions.

As shown in Fig. 7.12, **FixedRTX** significantly improves the trade-off between DMR and BWC against existing baselines while **FixedSame** cannot. This demonstrates that discriminately adding FEC over initial transmission and retransmission packets can effectively improve performance. As we discussed in §7.3, even by naively discriminating the retransmissions with another fixed redundancy rate would already be helpful, illustrating the necessity

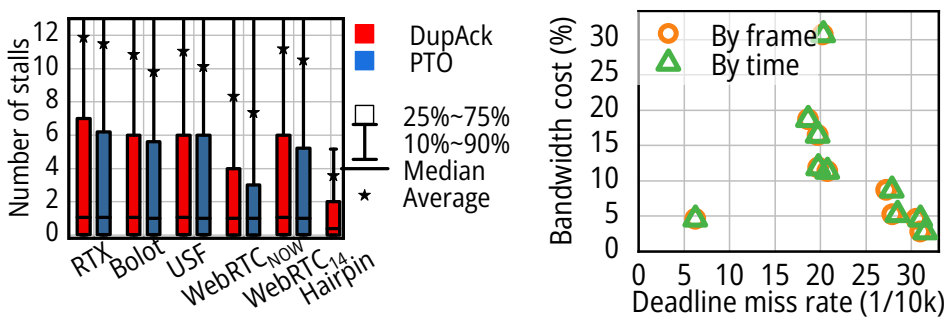


(a) Number of rounds. (b) Loss rates by round.

Figure 7.13: The loss rate in each transmission round.

of discriminating retransmissions. **Hairpin-Same** even behaves worse than the **FixedRTX** baseline, demonstrating the harm of adding redundancy to the initial transmission packets. Nevertheless, Hairpin still outperforms **Hairpin-Same** by reducing BWC by half (targeting the same DMR), demonstrating the optimality of the Markov-chain model.

Understanding Hairpin's decisions. In Appendix D.4, we present the redundancy rate and block size results of Hairpin to provide a deeper understanding of how Hairpin optimizes in different scenarios. Besides, we present the number of transmission rounds of Hairpin and baselines in Fig. 7.13(a). When Hairpin gradually increases the redundancy rate in future transmission rounds, most frames could therefore be delivered. Thus, the 99.9th percentile of the number of transmission rounds in Hairpin is less than all other baselines by more than one. Similarly, when we inspect the loss rate in each round as shown in Fig. 7.13(b), Hairpin also successfully maintains the lowest loss rate when the transmission round goes up. Note that the loss rate here is significantly high due to the survivorship bias – only lost packets will have another transmission round, while loss has already indicated a degraded network performance. This also indicates that the loss is not i.i.d. but bursty in the experiments.



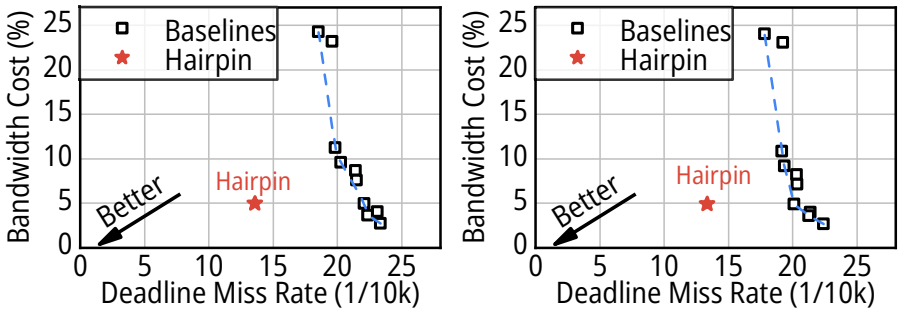
(a) Number of stalls per session.

(b) DMR by frame and by time.

Figure 7.14: The effect of DMR on other metrics.

Optimizing towards extremely low DMR. We further illustrate why we need to achieve an extremely low DMR and how it affects user's experience. As analyzed in §7.3.1, a lower DMR approaching zero directly indicates fewer stall events in a gaming session. We measure the number of *stall events* in each gaming session, where stall event is only counted once if there are multiple missed frames in one second or if it lasts longer than one second. As shown in Fig. 7.14(a), Hairpin can reduce the average and median number of stall events (which is also critical for user's opinion scores [225]) by a half or more against baselines. By having a DMR of 0.06%, Hairpin is able to reduce the 75th percentile number of stalls in a session to 2. Considering the duration of a gaming session (minutes to hours), this will considerably improve the user's experience.

We also show the difference of calculating DMR *by frame* and *by time* in Fig. 7.14(b). In this paper, we do not argue using a new metric (DMR by frame) is better – we calculate DMR by the number of missed frames over total frames because of the simplicity in the formulation in §7.3.3. Calculating DMR by time is almost equivalent to DMR by frame since the stalled time is the number of stalled frames (missed frames) times the interval be-



(a) GCC [279]. (b) NADA [289].

Figure 7.15: The performance of Hairpin and all baselines (labels omitted for brevity) on WiFi traces with different deadline requirements.

tween frames. Therefore, we replot Fig. 7.8(b) using two different DMRs. As shown in Fig. 7.14(b), the results are almost the same with each other.

Integrating with congestion control. To further investigate the performance of Hairpin when interacting with the CCA, we integrate the Hairpin with two CCAs in the WebRTC framework, GCC [77] and NADA [289], in our simulation. We then replay the collected traces by setting their bandwidth, RTT, and loss rate to the link in ns-3. The bandwidth ranges from 2Mbps to 30Mbps. As shown in Fig. 7.15, Hairpin could still achieve significant advantages over all existing baselines.

7.4.6 Real-World Experiments

Finally, we deploy the Hairpin in a production server in our cloud gaming service. We conduct an A/B test in production of Hairpin against the $\texttt{WebRTC}_{NOW}$ baseline. The bit rate of the cloud gaming service also supports the range of 2-30Mbps as simulated in §7.4.3. The A/B test runs for one week in September 2021, covering 17k sessions in total, all of

Ethernet	DMR	BWC	P(DMR>1%)	#Session
WebRTC$_{NOW}$	0.34%	30.4%	6.9%	8380
Hairpin	0.23%	3.0%	4.6%	7306
WiFi	DMR	BWC	P(DMR>1%)	#Session
WebRTC$_{NOW}$	0.72%	31.8%	19.3%	652
Hairpin	0.51%	3.0%	15.3%	613

Table 7.2: Real-world experiment results. P(DMR>1%) denotes the ratio of sessions with an average DMR of larger than 1%.

which have a duration of at least 4 minutes. Hairpin has been integrated into the UDP-based connections of our cloud gaming service since then. Since other optimizations are also deployed into our service after we deploy Hairpin, to make a fair comparison, we only present the results from the controlled A/B test in September 2021.

Performance. As shown in Table 7.2, Hairpin is able to improve both the average DMR and the average BWC compared to WebRTC$_{NOW}$. Specifically, for Ethernet sessions, Hairpin could improve the DMR by 32% while also reducing the BWC by 40% against WebRTC$_{NOW}$. For WiFi sessions, the improvements on DMR and BWC are 30% and 43%. We also measure the ratio of sessions with an average DMR of larger than 1%, i.e. tail sessions. Hairpin could also reduce the tail sessions by 34% and 21% for Ethernet and WiFi sessions respectively compared to WebRTC$_{NOW}$. Note that the DMRs in real-world experiments are a little higher than those in simulations (§7.4.3). This might be because of other external factors (e.g., user devices) that could affect the DMR. Nevertheless, Hairpin could significantly improve the users' experiences on both the DMR and BWC compared to WebRTC$_{NOW}$.

Overhead. We further measure the overhead of the optimization of Hairpin. As introduced in §7.3.4, to accelerate the optimization online, we precompute the optimized FEC pa-

rameters and store the result table for online look-up. At our quantization granularity of the table, it takes 1.98MB to store the table, which is negligible on servers since the table is static and could be shared by all connections. Moreover, according to our measurements, the time of looking up the table is always less than 1ms, which is also negligible since the table is looked up at the granularity of the frame.

7.5 Limitations

Delay components in interactive streaming. Hairpin could have maximum benefits when the end-to-end network delay dominates the total delay from the video encoder to the decoder in Fig. 7.9. This is generally true in interactive streaming services. Related measurement studies also demonstrate that the network delay is still one of the bottlenecks of edge-based interactive streaming [124, 190]. Therefore, we focus on the optimization of streams between edge servers and clients. Our deployments in the wild demonstrate that optimizing the network latency could significantly improve the user's experience (note that DMR is measured end-to-end). Hairpin can also work with the optimization of other delay components (e.g., encoding, decoding, etc.) to further improve the performance.

Deployment efforts for applications. Another concern of deploying Hairpin is that both the server and the client need modification to support the redundancy and retransmissions. There are previous efforts implementing the FEC mechanism over TCP [54, 111], which needs to modify the TCP protocol stack at the client and are not suitable for products at scale. For scenarios where TCP is compulsory for transport, the deployment of Hairpin may depend on the ability to modify the reception mechanism of TCP packets at the client.

However, most interactive streaming applications adopt UDP to reduce the network delay [67, 96, 182, 224], including our service. In this case, Hairpin could be implemented within the application at the server and the client, which is practical for most applications.

7.6 Summary

We propose Hairpin, a packet loss recovery mechanism for edge-based interactive streaming to jointly optimize redundancy and retransmissions. Hairpin motivates the joint optimization with real-world measurements, and optimizes the redundancy and retransmissions with Markov decision process. Both trace-driven simulations and real-world deployments show that the joint optimization significantly reduces the DMR and BWC compared with state-of-the-art solutions.

8

Network Layer on Data Path: Smooth Queue Management

8.1 Introduction

In-network packet scheduling and queue management are powerful tools to ensure that competing networked applications fairly share network resources and achieve their perfor-

mance objectives (i.e. high throughput, low latency) as best possible. However, emerging real-time streaming applications such as video conferencing, online gaming, and virtual reality suffer from *performance volatility*. Performance volatility manifests as sudden, abrupt drops in throughput or spikes in latency, often as a result of bursty arrival patterns of competing traffic. Performance volatility results in glitches and stalls for applications with *heavy, real-time* (HRT) traffic* (such as video conferencing). Indeed, prior work shows that a latency spike of only 200 ms can lead to several seconds of recovery time at the application layer [188].

Troublingly, we observe that many advanced queueing disciplines today not only fail to prevent performance volatility but that they actually *aggravate* volatility. The problem stems from a fundamental tension between two desirable properties: maximizing throughput fairness and minimizing performance volatility. *We observe that strict fairness entails high volatility in the presence of bursty workloads, and that naively mitigating volatility entails weakening fairness.*

To understand the crux of the conflict between fairness and volatility, we consider a motivating example in Figure 8.2(a). An HRT video connection runs alone over a residential network link, when another user loads a web page (namely, `amazon.com`, settings in §8.7.1). In experiments with a range of queueing disciplines, we see that the video connection experiences an unacceptable (>190ms [216]) frame delay lasting for as much as a second. On the one hand, the worst-performing queueing discipline for the HRT flow is *fair queueing* (FQ), which benefits fairness. Indeed, FQ rapidly shifts bandwidth resources to the new

*HRT represents flows demanding high throughput and low latency at the same time. For example, beyond requiring low latency, videoconferencing applications will also try to increase the bitrate for better quality [163].

Web flows, bottlenecking the HRT flow, which will require several RTTs before it receives adequate signals to adjust its video bitrate and its congestion window. On the other hand, the best setting among existing schemes for the HRT flow is the *least fair* one since it simply benefits the HRT flow at the expense of the Web traffic.

An intuitive solution to the volatility vs. fairness tradeoff might involve some sort of priority scheme with surgically computed 'weights' to prioritize sensitive classes of traffic to avoid extreme unfairness. Unfortunately, this is impractical. First, labeling flows (e.g., with DSCP bits [53]) in this way is *not* incentives compatible [†] since Internet senders would always benefit from labeling their traffic with higher-priority classes. Worse yet blindly adhering to potentially buggy labeling of various applications will immediately deprive us of any performance guarantee. Second, administrators cannot simply assign weights of classes a priori, because traffic distribution is dynamic and largely unpredictable.

The above discussion leads us to our quest for a queue management scheme that balances three properties that lie in tension with each other. First, we desire a scheme which, in the long run, adheres to traditional flow-rate fairness. Second, we desire a scheme that tames volatility and enables HRT flows to live side-by-side with bursty traffic patterns (namely, web traffic). Finally, we desire a scheme that is practical, in the sense that it is parameter-free like CoDel [203] and does not require any flow labeling by senders or application-specific configurations such as deadlines [82].

To this end, we designed Confucius[‡], a parameter-free queue management scheme that balances fairness versus volatility. In the long run, Confucius guarantees fair flow schedul-

[†]Recent efforts (e.g., L4S [69]) which use incentives-compatible labeling still suffer from practicality and performance issues, as we will later show.

[‡]One of Confucius' (the philosopher) educational philosophy is teaching students according to their needs, where in this paper we are going to serve the flows according to their needs.

ing between competing classes of traffic. However, in the short run, Confucius refuses to abruptly adjust service rates upon bursty traffic arrivals. Instead, when new flows arrive and service rates must be adjusted to ensure fairness, Confucius*gradually* adjusts the weights to provide HRT flows a few RTTs to detect the change in network conditions and adjust their bitrates and congestion windows appropriately. More specifically, Confucius assigns flow rates according to a simple exponentially weighted moving average (EWMA [175]) which *smoothly* moves rates towards a fair allocation. We find that this approach provides a good tradeoff between fairness and volatility; in experiments, we measure flow-completion times (FCTs) for web traffic (which benefit from strict fairness) versus frame delays for HRT flows (which benefit from smoothing) to understand the impact of this tradeoff. In trace-driven experimental tests, we find that Confucius typically reduces the duration of frame delay degradation of HRT flows by 90% while maintaining comparable FCTs for web traffic.

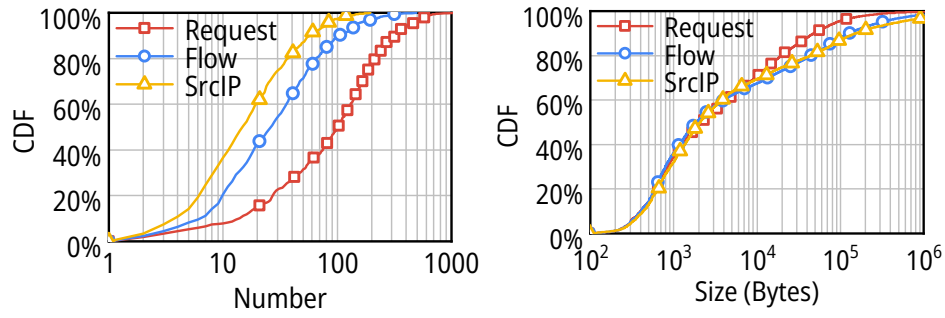
We faced several challenges in designing Confucius:

Practicality Confucius is a classful queueing scheme, which (like many other classful schemes [231, 240]) groups low-latency flows into the same queue to avoid the latency impact of sharing a queue with buffer-filling traffic. This begs the question of how Confucius can be parameterless, correctly classifying flows without the use of labels. In §8.5, we illustrate how Confucius adaptively migrates flows between classes depending upon their queue occupancy: flows that naturally occupy a small fraction of the buffer are clustered together, while flows that are observed to be buffer-filling compete in a shared buffer with other buffer-filling flows.

Performance Guarantees It is easy to vaguely describe Confucius as 'balancing fairness and volatility' but it is harder to formulate this into a rigorous service model. By mathematically analyzing the EWMA function which Confucius uses to adjust service rates, we calculate performance bounds for a few classes of applications that might use Confucius. We show that short, FCT-driven flows (such as web traffic) observe a maximum slowdown of 360 ms relative to fair queueing in our setting; HRT flows (such as real-time video) experience more than 90% less stalls compared to fair queueing, and that long-lived, bulk transfers experience no degradation at all relative to fair queueing (in the limit).

Avoiding Oscilations Enforcing fairness and consistency in a dynamic environment with multiple control systems (e.g., congestion control, bit-rate adaptation) operating concurrently is dangerous. Seemingly minor changes in queue management could have large collateral damage to applications. By jointly and cautiously assigning the service rate per queue and the flows per queue, Confucius avoids conflicting decisions that will be detrimental to stability. More importantly, Confucius's control is strategically slow-moving, effectively leaving enough time for other control systems, especially congestion control, to kick in to react optimally.

Before moving forward, we consider one issue of setting. Confucius is designed for deployment in residential and end-user access points (e.g., WiFi APs or cellular base stations), and our experiments and data involve application use in those settings where it is well-known that congestion is frequent [52, 117, 188]. There is an open discussion in the networking community in exploring congestion's impact in other settings (e.g., in the Internet core [95] or in datacenters [51]), but these other settings are out of scope for Confucius.



(a) Connections per Web page. (b) Size distributions.

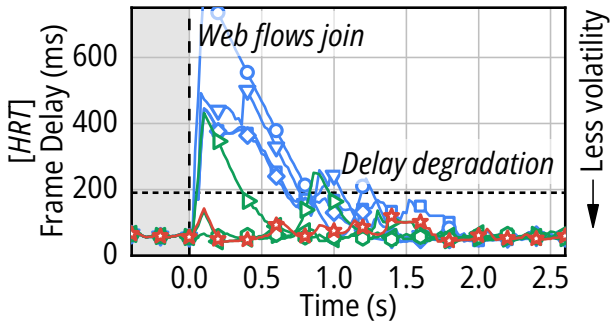
Figure 8.1: Number of TCP flows and their size for loading each of Alexa top 1000 websites (measure time: July 2022 from one vantage point with Chrome and capture the HAR log [1].

Moreover, the computation capability at edge routers also enables us to fine-grained traffic management for flows, as we will demonstrate in §8.7.5.

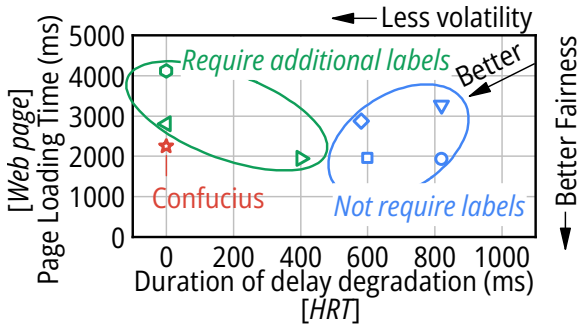
8.2 Motivation

We start by describing recent trends in Internet applications that call for reconsidering queue management. Next, we explain via an intuitive running example why existing approaches in both AQM and scheduling fall short in addressing these challenges.

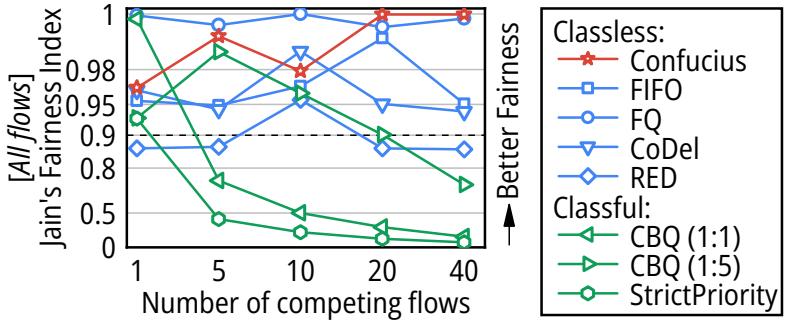
The rise of HRT brings new challenges to queue management. While the Internet always carried multiple applications, the emergence of prosperous real-time communication applications (e.g., videoconferencing, cloud gaming, virtual reality), in particular, has made sharing of bottleneck links particularly challenging. HRT applications require not just low latency but consistently low latency while also sending at very high bitrates [163, 188]. Despite recent advances in wireless technologies such as 5G and WiFi 6 [62, 188, 264], the HRT consistency requirement is often violated, bringing bad user experience. Facilitating



(a) The HRT flow's latency over time.



(b) The HRT vs. Web flows degradation.



(c) Jain's fairness index (JFI) as the workload changes.

Figure 8.2: (a) A pre-existing HRT flow (e.g., videoconferencing) competes with flows of a Web-page load (namely, `amazon.com`). The HRT flow experiences transient delay degradation with classless (blue) schemes, while Web traffic experiences long page load times with classful (green) schemes. **(b)** Each scheme manages a different balance between the HRT (volatility) and web traffic (fairness). **(c)** The fairness of classful solutions (e.g., CBQ) is heavily sensitive to workload variations. For instance, CBQ with different weights (1:1 or 1:5) will result in poor fairness (JFI$<$0.9) in certain workloads. Y axis is not lin-scaled.

the HRT consistency objectives requires queue management schemes to shift from preventing *fairness* to also preventing performance *volatility*.

Volatility is very hard to avoid in the Internet. While intuitively, providing consistent performance in the Internet could be addressed by recycling good old AQMs, two key characteristics make this task particularly challenging. First, Internet traffic is often bursty. As an intuition, a simple page load results in a burst of responses from multiple sources. In fact, the median number of flows that a webpage load generates is 27, while for 25% of websites that number is 56 flows. As an illustration, we present the number of HTTP requests, concurrent flows (defined by 5-tuples), and source IPs in Fig. 8.1(a). Second, while most AQMs schemes were designed with loss-based CCAs in mind, today's applications run multiple distinct congestion control algorithms in accordance with their distinct objectives. Importantly, ten distinct algorithms are used by the top Alexa websites [193].

Research Question. Taken together, these trends beg the question: Are today's in-network queue mechanisms (i.e. AQM and scheduling) able to *fairly and consistently* satisfy the heterogeneous objectives of flows sharing a bottleneck link while being practical?

8.2.1 Motivating Example

To answer this question, we present an intuitive experiment. Assume a user has a video call, thus pulling an HRT (*heavy, real-time*) flow through a router. At t=0s, another user opens a web page and creates a burst of new short flows on the same bottleneck link as the video flow. The two applications use different CCAs, to achieve their objectives. Concretely, the HRT flow uses Copa [47], a low-latency CCA for videos [122] and the webpage uses

TCP Cubic. Fig. 8.2 illustrates the experience of the two applications (a) over time, (b) on average, and (c) in terms of fairness (JFI), when the bottleneck link is controlled by a variety of schemes. We explain the experimental settings in more detail in §8.7.2. While simple, our example practically demonstrates the tension between fairness and volatility. Thus, the observations we draw from this example generalize to other traffic mixes and scenarios as we show in §8.7.

We distinguish existing schemes in *classful* and *classless*. The former requires end-hosts to label packets per application (videoconferencing, or web). The latter does not need or leverage end-host labels.

Unfortunately, none of the existing solutions can adequately address the tension between fairness and volatility in a realistic setting. Specifically, these existing solutions, respectively, have one or multiple of the following issues:

Performance volatility: the HRT flow suffers from delay degradation when Web flows join. Classless schemes such as FIFO, FQ, RED are unable to avoid performance volatility, effectively hurting the HRT flow. As we observe in Fig. 8.2(a), when classless schemes (in blue) are managing the bottleneck link, the HRT flow experiences high delays. Concretely the delay of HRT increases by 4× reaching 400-800 ms. In perspective, an end-to-end delay for video frames of more than 190 ms (dashed line in Fig. 8.2(a)) causes a stall in video streaming [216]. Fig. 8.3(a) and 8.3(b) visually explain why simple classless schemes such as FQ and FIFO are so bad at avoiding volatility. Observe that the available bandwidth for the HRT flow reduces so abruptly when the web flows arrive that the HRT flow cannot adapt.

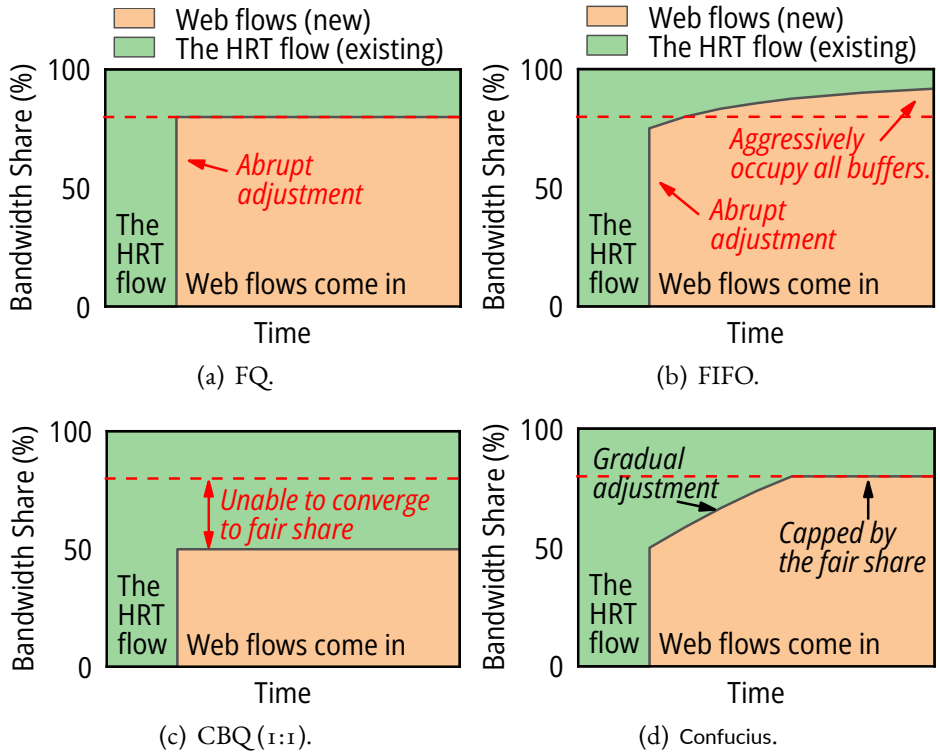


Figure 8.3: Illustration of how bandwidth shares change over time with incoming flows for different scheduling algorithms. The dashed red line marks the fair share of the HRT flow.

Failing to offer consistent latency is an unintuitive result for AQM schemes that actually try to control end-to-end latency [79, 109, 113, 203]. However, traditional AQMs cannot balance the performance of heterogeneous flows, as they were designed with loss-based CCAs in mind [107] and cannot effectively communicate congestion to delay-based CCAs, which are adopted by most real-time flows [77]. For multiple latency-sensitive CCA's (including GCC and Copa), a sender does not interpret AQM-induced losses or ECNs as congestion, thus would not reduce its sending rate until the loss rate is very high. Therefore, as shown in Fig. 8.2(a) and 8.2(b), AQMs such as CoDel and RED result in significant delay degradation for the HRT flow.

Unfairness: either the Web flows or the HRT flow suffer from extreme performance degradation. Classful schemes such as CBQ, which splits packets into classful queues of configurable service rate or strict priority, which only dequeues packets of lower priority if high priority is empty, protect the HRT flow, as we observe in Fig. 8.2(a). However, classful schemes also result in unfair allocations (as shown in Fig. 8.2(b)) because they overpenalize (or even starve) web traffic which experiences high page load times (PLTs) as shown in Fig. 8.3(c). While, in theory, CBQ could be configured to be fair, that requires knowledge of the exact workload (ratio of flows between classes) over very short time intervals, which is in practice infeasible. For example, we measure the fairness that different schedulers can provide while changing the number of competing flows to the HRT flow in Fig. 8.2(c). Modifying CBQ's configuration improves JFI for a subset of the workloads: CBQ (1:1) works well when there are two flows competing while CBQ (1:5) achieves a good JFI when there are five competing flows – they both degrade as the number of flows changes.

Impracticality: requiring end-hosts to correctly label their traffic is unrealistic in the Internet. Besides the sensitivity to configuration, classful schemes require the end-host to label flows according to their importance or objectives and prioritize traffic based on that. Such label-driven management is unrealistic for home routers for the following reasons. First, labeling incurs substantial coordination overhead. Indeed, users will need to use labels according to their application objectives while also agreeing with routers on the meaning of these labels. Second, label-driven management assumes end hosts are trusted and bug-free. In practice, senders have the incentive to label their flows with a higher prior-

ity. Thus, such schemes are mostly practical only for datacenters where both end-hosts and routers are under the control of the same entity (e.g., LSTF [194], pFabric [42]).

While simplistic, our motivating example teaches us two lessons about how we should treat flows of various objectives or CCAs:

Takeaway 1. *Immediately enforcing bandwidth fairness e.g., upon arrival of a traffic burst, hurts the performance of existing flows due to the disparity between the CCAs' sending rate and the available bandwidth in the bottleneck link. CCAs might not have information about the dramatic decrease in bandwidth early enough to react gracefully.*

Takeaway 2. *Flows driven by different CCAs or having distinct objectives should not share the same queue because their perception of congestion differs. As a result, even advanced AQM schemes cannot signal congestion in the right way and at the right time for each of them independently.*

8.3 Confucius Design

In this section, we explain how the takeaways from §8.2 manifest in the design of Confucius, a scheme that pushes forward the Pareto frontier between fairness and non-volatility. To this end, we explain how Confucius re-allocates bandwidth upon arrival of a burst of new flows to avoid performance volatility. Then, we explain how Confucius splits bandwidth across new and existing (old) flows to achieve equitable performance (fairness).

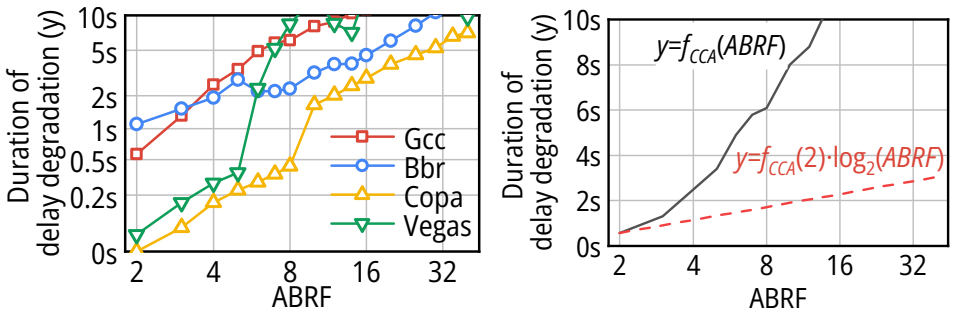
8.3.1 Taming volatility through cautious bandwidth re-allocation

To address the performance-volatility problem Confucius leverages a simple yet powerful insight that stems from Takeaway 1: Upon the arrival of a burst, the reduction of the bandwidth that is available to existing (old) flows is inevitable if we want to preserve long-term throughput fairness. Yet, if we gradually and cautiously control the reduction of the bandwidth during the transient period, we can eliminate the disparity between the sending rate of the old flows' CCA and the actual service rate at the bottleneck link, thereby taming volatility.

To understand why there is an advantage in *gradually* controlling the HRT flow's bandwidth allocation compared to directly cutting its available bandwidth to its fair share, we measured the duration of severe delay degradation y. Concretely, y denotes the time interval during which an HRT flow would experience a delay of more than 190 ms of delay[§]. We plot y as a function of the *Available Bandwidth Reduction Factor* (ABRF) for different CCAs in Figure 8.4(a). We find that CCAs respond very poorly to sudden, large reductions in bandwidth. For instance, reducing GCC's available bandwidth to one-sixteenth of its initial value (i.e., $ABRF = 16$) results in $y > 10$ *seconds* stalls of video frames. Interestingly, we observe in Fig. 8.4(a) that the curve $y = f_{CCA}(ABRF)$, as we denote the relationship between the ABRF and the duration of delay degradation y, follows a super-linear relationship.

To avoid such delay degradation, Confucius *gradually* reduces the available bandwidth for the HRT flow. For instance, to achieve a final ABRF of 16, one might use $log_2(16) = 4$ iterations of bandwidth reduction if the weight is smoothed. Such an exponential (smooth)

[§]This is the recommended network delay for video chats by ITU [216]



(a) Measurements (b) Illustration

Figure 8.4: (a) Duration of delay degradation increases with the available-bandwidth-reduction factor (ABRF). (b) An illustration of how gently reducing available bandwidth helps reduce delay duration. Note that (a) is a log-log plot but (b) is a log-lin plot.

change in bandwidth share can be achieved by using EWMA and cutting the HRT flow's bandwidth by half at each iteration. This would give the CCA an opportunity to learn about the reduced bandwidth allocation through its usual congestion signals while simultaneously mitigating the disparity between the flow's sending rate and available bandwidth at every iteration, thus taming volatility. Figure 8.4(b) demonstrates, in the ideal case, the value proposition of this approach: instead of scaling *super-linearly*, the duration of delay degradation increases only *logarithmically* with the ABRF (modulated by $f_{CCA}(2)$, a small constant).

Applying a logarithmic dampening factor to the HRT flow's available bandwidth (instead of an instantaneous reduction), Confucius no longer preserves *strict fairness*. Intuitively, that could result in severe damage to short flows. Yet, we prove in §8.4.2, that Confucius guarantees that the FCT for short flows will *always be within a constant, additive factor* of the FCT under a strictly fair allocation.

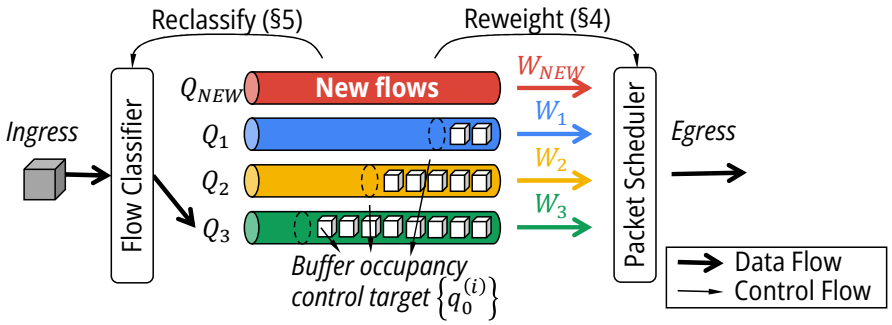


Figure 8.5: Design overview of Confucius. w_i denotes the weight for queue i in the scheduling with DWRR.

8.3.2 Equitable handling of competing flows

Having explained how Confucius gradually re-allocates bandwidth between old and new flows, we discuss how Confucius actually splits this bandwidth among individual flows. At a high level, Confucius first splits flows to queues and strategically assigns a portion of the available bandwidth to each of them, as illustrated in Figure 8.5.

Following Takeaway 2, splitting flows into different queues is essential and challenging. Indeed, putting all old flows in a single FIFO queue will lead HRT flows (e.g., Copa) to starvation [47] if flows use heterogeneous CCAs. But, using FQ to split old flows may not be able to provide low latency to the bursty old flows [176].

Confucius splits flows into queues according to their objectives on the premise that flows of similar performance objectives will not hurt each other. To identify the objective of flows in the system, Confucius uses the queue occupancy. *We find that flows implicitly demonstrate their preferences and objectives based on how they utilize the bottleneck queue.* For example, latency-sensitive applications will choose CCAs that can achieve low latency such as Copa [47] or GCC [77]. Such CCAs achieve low latency by trying to keep the

bottleneck queue as short as they can. In contrast, throughput-oriented CCAs (e.g., Cubic) will keep the buffer full to maximize the utilization for the throughput. This allows us to identify the latency preference of flows by their queue occupancy: if one flow has a low queue occupancy, it indicates that (i) that flow tries to not overutilize the queue; and (ii) that flow can co-exist with other flows with similar behaviors.

By grouping flows with similar queue occupancy into the same queue, flows with different queue occupancy will not affect each other. Meanwhile, with a fixed number of queues to schedule between, latency-sensitive flows, no matter bursty or not, will have a consistent latency. Thus, Confucius has a set of queues, each designed to accommodate old flows with different buffer occupancy, and a separate queue dedicated for short flows. Confucius adopts a Deficit-Weighted Round-Robin (DWRR) algorithm to schedule between these queues. When a new flow arrives at the router, Confucius will put it into the short-flow queue. Confucius will periodically measure flow characteristics and reclassify flows as necessary. Doing so allows Confucius to measure flow characteristics accurately. To further increase the robustness of the performance in practice, we introduce hysteresis-based mechanisms for the reclassification of flows. We elaborate on this mechanism in §8.5.

Having categorized flows according to their objective the next natural question is *(i)* how to split bandwidth across those categories; and *(ii)* how long to wait before changing the bandwidth allocation. For the former, our insight is that bandwidth allocation needs to depend on the ratio between the number of old and new flows. For the latter, our insight is to move bandwidth to new flows from old ones so fast as the old flows' CCA has time to react.

In practice, respecting the reaction time of each CCA means that we need to adapt the design of Confucius in various CCAs. To this end, we plot response curves for different CCAs and find that the reaction time of CCAs during bandwidth changes is always above a certain threshold, where Confucius always benefits from gentle adjustments. We could therefore design a uniform weight-adjustment algorithm for flows with different CCAs. In practice, Confucius effectively focuses on the least reactive CCA to make sure all CCAs can have adequate time to react.

8.4 Age-aware Flow Weights Adjustment

In this section, we dive into Confucius' weight adjustment (§8.4.1). We then analytically show that this mechanism guarantees bounded performance degradation, both for existing HRT flows and newly-arrived mice flows (§8.4.2).

8.4.1 Adjustment Mechanism

Recall that Confucius classifies flows into different queues and uses DWRR to schedule packets across these queues. To assign weights (i.e. service rates) to queues, Confucius uses the following process. For each flow, f, Confucius first computes a weight, w_f; then, for a given queue, Q, the weight is computed by summing up flow weights of all flows in Q:

$$W_Q = \sum_{f \in Q} w_f \tag{8.1}$$

A key ingredient in Confucius' design is the computation of per-flow weights. For this purpose, Confucius distinguishes *new flows* from *old flows*. In fact, Confucius groups new

flows into a separate queue called Q_{new} (depicted in Figure 8.5). All old flows which are mapped to Q_1, Q_2,..,Q_n are assigned a flow weight of $w_f = 1$, and are collectively designated by the set $\mathcal{F}_{old}$. When a new flow arrives, it is first mapped into Q_{new}, and the flow weights of all flows in Q_{new} are then recomputed as follows:

$$w_f = \min\left(\frac{|\mathcal{F}_{old}|}{|Q_{new}|} \cdot 2^{\lambda t},\ 1\right), \quad f \in Q_{new} \tag{8.2}$$

There are several considerations in the design of Eq. 8.2:

Age-aware adjustment $\left(2^{\lambda t}\right)$. As described in §8.3.1, Confucius *gradually reduces* the available bandwidth for HRT flows. To achieve this, Confucius *gradually increases* the weights of the competing new flows. Here, t represents the age (in milliseconds) of the new flow, and λ is a parameter that controls the rate at which the flow weights for mice flows are adjusted – flow weights double every $\frac{1}{\lambda}$ milliseconds. The higher λ is, the faster new flows converge to their fair share of the bandwidth, and the more abrupt the reduction in available bandwidth for HRT flows. We will discuss how λ affects the performance degradation quantitatively in §8.4.2.

Initial weight $\left(\frac{|\mathcal{F}_{old}|}{|Q_{new}|}\right)$. If the initial flow weight for new flows is too small, even an exponential growth factor would result in a protracted convergence period for these new flows. In particular, when there are already many old flows, it is hard for few new flows to grab their fair share of bandwidth. Therefore, we scale the initial weight of new flows with the *number of old flows* that are currently active in the router. For each new flow, we set the initial weight to $\frac{|\mathcal{F}_{old}|}{|Q_{new}|}$, where $|\mathcal{F}_{old}|$ and $|Q_{new}|$ are the total number of old and new flows, respectively. The intuition behind this particular choice of initial weight is always limiting

the bandwidth reduction for old flows to be less aggressive than a factor-of-2 reduction. In this case, the duration of delay degradation can logarithmically scale with the base of $f_{CCA}(2)$, as shown in Figure 8.4(b).

Upper bound $(\min(..., 1))$**.** Confucius uses a flow weight threshold of 1 to 'age out' new flows from the Q_{new} queue. Once the flow weight of a flow reaches 1, the flow is no longer considered new, and is moved to one of the other queues based on the output of the Flow Classifier (§8.5).

Parameter configuration. The choice of λ is an important consideration in the design of Confucius. A large λ (e.g., $\lambda \rightarrow \infty$) leads to abrupt reductions in available bandwidth, causing volatility, while a small λ (e.g., $\lambda \rightarrow 0$) results in unfairness (or even starvation) for new flows. Moreover, in setting this parameter, we need to be aware that different flows, particularly flows with different CCAs, respond differently to the same congestion signals (e.g., Copa requires 5 RTTs to effectively reduce its sending rate, while BBR's response time is dictated by its probing interval of 6-8 RTTs). Consequently, we seek to configure λ so that the available bandwidth drops as fast as possible, subject to the responsiveness of the underlying CCAs.

To deal with the heterogeneity of CCAs on the Internet [193], we set λ as the inverse of *the probing period of the least responsive, latency-sensitive CCA*. This ensures that even the least responsive CCA can smoothly react to bandwidth changes. Based on the experiments depicted in Figure 8.4(a), BBR is the least responsive CCA with a probing period of 6-8 RTTs. Therefore, given a typical RTT of 30-50 ms for Web services [276], we set λ=0.004

Parameters and variables:	
B	Size of each new Web flow.
N	Number of new Web flows.
k	The responsiveness of a CCA.
q_0	The delay target that a CCA will try to achieve.
C	The link capacity.
τ	The feedback loop of a CCA (usually one RTT).
B_0	The initial burst of a new flow (e.g., the initial cwnd [102]).
P	The scheduling policy.
Functions:	
$s(t)$	Sending rate of the HRT flow of time t.
$r(t)$	Available bandwidth of the HRT flow of time t.
$p(t)$	Number of packets in the queue of the HRT flow.
$q(t)$	The queueing delay of the HRT flow.

Table 8.1: Notations

(ms^{-1}) to have a doubling interval of $\frac{1}{\lambda}$=250 ms. Experiments in §8.7.2 demonstrate satisfactory results for not only BBR but also several other CCAs.

8.4.2 Theoretical Analysis

In this subsection, we demonstrate analytically that Confucius can provide consistent performance for both HRT and Web flows. In the scenario of a single HRT flow competing with N new flows (e.g., Figure 8.2(a)), we show that Confucius guarantees bounded delay degradation for the existing flow, while yielding FCTs for Web flows that are within a constant additive factor of what FQ provides. We list the notations we will use in Table 8.1.

Scenario overview. Consider a single HRT flow running by itself on a bottleneck link. At $t = 0$, N new flows (e.g., Web flows), each with size B, join the same bottleneck link and

share the buffer with the existing flow. We analyze the performance degradation for both the existing and new flows.

CCA model. We adopt a simplified delay-convergent CCA model [41, 48], where the delay-sensitive CCA has a target queueing delay, q_0. The CCA seeks to maintain its queueing delay around this target, increasing or decreasing its sending rate proportional to the difference between the current delay and the target:

$$\frac{\mathrm{d}s(t)}{\mathrm{d}t} = -k \cdot (q(t-\tau) - q_0) \tag{8.3}$$

Here, $s(t)$ is the flow's instantaneous sending rate, $q(t)$ the instantaneous queueing delay it experiences, and τ is the feedback loop of the CCA. Finally, k is a coefficient representing the CCA's responsiveness. We discuss how k varies for different CCAs in Appendix E.1.5.

Delay model. Next, we analyze the number of packets in the queue, $p(t)$, at time t. At any $t > 0$, this quantity satisfies the following relationship:

$$p(t) = p(0) + \int_0^t (s(t') - r(t'))\,\mathrm{d}t' \tag{8.4}$$

where $p(0) = q_0 \cdot C$ is the buffer occupancy in steady state with C being the link capacity. If $r(t)$ represents the instantaneous service rate (i.e. available bandwidth) for the HRT flow at time t, then the queueing delay can be written as follows:

$$q(t) = \frac{p(t)}{r(t)} = \frac{1}{r(t)}\left(p(0) + \int_0^t (s(t') - r(t'))\,\mathrm{d}t'\right) \tag{8.5}$$

There are two metrics that we focus on. The first is the **maximum queueuing delay** experienced by the HRT flow, q_P^{max}, for a given scheduling policy P:

$$q_P^{max} = \max_{t>0} q(t) \tag{8.6}$$

In this context, we find that q_P^{max} serves as a good proxy for the duration of delay degradation since it establishes a *lower bound* on how quickly previously-queued packets of the HRT flow drain from the bottleneck queue.

The second metric is the **FCT**, T, for the new flows, which can be expressed as follows:

$$\int_0^T (C - r(t'))\,\mathrm{d}t' = N \cdot B \tag{8.7}$$

Since FQ provides the 'fairest' bandwidth allocation (representing one extreme of the fairness vs. non-volatility tradeoff), we use the FCT for Web flows under FQ, T_{FQ}, as our baseline. We then calculate $T_P - T_{FQ}$ as the degree to which policy P degrades Web flow performance relative to FQ.

Having established our two figures of merit (*maximum queueuing delay* and *FCT degradation to FQ*), we evaluate four scheduling policies: FQ, FIFO, CBQ (1:1), and Confucius.

Policy P	q_P^{max}	$T_P - T_{FQ}$
FQ	$\approx N\left(\frac{2}{3}\sqrt{\frac{2}{k}} + q_0 + \tau\right)$	0
FIFO	$\approx \left(\frac{NB_0}{q_0 C} + 1\right)\left(\frac{2}{3}\sqrt{\frac{2}{k}} + q_0 + \tau\right)$	$\lessapprox 0$
CBQ	$\approx \frac{2}{3}\sqrt{\frac{2}{k}} + q_0 + \tau$	$\approx \frac{(N-1)B}{C}$
Confucius	$\approx 6q_0 + 15\tau + \frac{8\lambda}{k} + \frac{(10q_0+15\tau)\lambda^2}{k}$	$\approx \frac{\log_2 e}{\lambda}$

Table 8.2: Approximations for different schedulers on their maximum delay (q_P^{max}) and FCT degradation ($T_P - T_{FQ}$). In the transient scenarios, existing scheduling policies have either unbounded delay degradation, or unbounded flow completion time degradation. The unbounded terms with workload changes (N and B) are marked in red.

We find that the available bandwidths for these policies satisfy the following relationships:

$$r_{FQ}(t) = \frac{C}{N+1} \qquad (t > 0) \tag{8.8a}$$

$$r_{FIFO}(t) \leqslant C \cdot \frac{Cq_0}{Cq_0 + NB_0} \qquad (t > 0) \tag{8.8b}$$

$$r_{CBQ}(t) = \frac{C}{2} \qquad (t > 0) \tag{8.8c}$$

$$r_{\text{Confucius}}(t) = \max\left(\frac{C}{2} \cdot 2^{-\lambda t}, \frac{C}{N+1}\right) \qquad (t > 0) \tag{8.8d}$$

where for FIFO, B_0 is the initial burst size of these new flows (e.g., the initial congestion window in TCP). We then solve for the performance degradation of the HRT flow, q_P^{max}, and FCT degradation of mice flows, $T_P - T_{FQ}$, with the differential equation in Eq. 8.5 using Laplacian transforms. We summarize the *approximate* results in Table 8.2 and leave the analytical details to Appendix E.1.

For FQ and FIFO, we observe that the duration of delay degradation scales linearly with the number of new flows, N, and is therefore unbounded, where N can go to more than 100 in some Web pages (Fig. 8.1(a)). Intuitively, as the number of flows joining the bottle-

neck link increases, the more drastically the available bandwidth for the HRT flow drops, resulting in significant volatility.

In the case of CBQ, pre-labelling the HRT flow enables the policy to give it a fixed share of bandwidth, resulting in bounded delay degradation. However, if the weights are not appropriated precisely (i.e., do not match the number of flows in each queue), CBQ converges to an unfair solution, and the degradation in FCT for mice flows becomes unbounded (§8.2).

Finally, Confucius yields bounded performance degradation for *both sets of flows*. On one hand, Confucius ensures that the delay degradation for HRT flows is a constant that depends only on the CCA's queueing delay target (q_0), the responsiveness of the CCA (k), the duration of its feedback loop (τ), and the decay parameter (λ)[¶]. On the other hand, Confucius can also ensure the FCT degradation for mice flows is bounded by an additive constant factor with respect to the decay parameter (λ), which goes to negligible with the increase of the flow sizes.

8.5 Occupancy-aware Flow Classification

As described in §8.3.2, Confucius seeks to classify flows into groups, each with a dedicated queue based on how aggressively they consume buffer space. In this section, we first present our design consideration when classifying flows into different queues (§8.5.1). We then present our hysteresis-based mechanism to robustly classify the flows (§8.5.2).

[¶]In practice, when using Copa with an RTT of 40ms, the approximation bound $q^{max}_{\text{Confucius}}$ from Table 8.2 is roughly 640ms. As we show experimentally in §8.7.2, the delay degradation using Confucius is much lower than this.

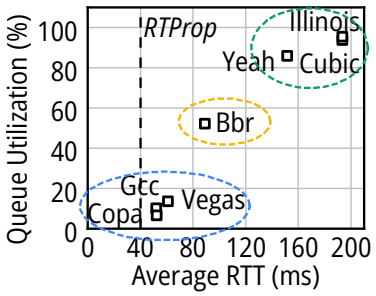


Figure 8.6: The relationship between queue utilization and delay in different CCAs. Experiments are simulated with real WiFi traces from [188].

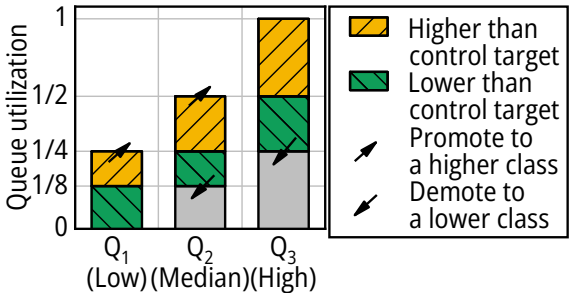


Figure 8.7: Confucius's hysteresis reclassification mechanism for flows. Only when the buffer occupancy of a flow has significantly deviated from the current class will it be moved to another class.

8.5.1 Design Considerations

Confucius puts short flows into a separate queue Q_{new} and classifies long flows with different buffer occupancy aggressiveness into separated queues. Therefore, we need to set up a series of queues $Q_1, Q_2, \cdots, Q_n$ to accommodate flows with different buffer occupancy.[‖] Queue indices increase with buffer target i.e. Q_1 will be shorter than Q_3, as shown in Fig. 8.5. Specifically, we denote the buffer occupancy that queue Q_i targets as $q_0^{(i)}$. Realizing this brings with two questions. First, how many queues we should set for routers to accommodate heterogeneous flows. Second, how to match the flow's buffer occupancy with the target $q_0^{(i)}$ that queue Q_i tries to maintain. We will answer these two questions in the following.

Number of queues to set. The first thing to determine for instantiating Confucius is how many queues we should set on the router. To answer this, we need to estimate how many CCA groups of distinct queue behavior there are in the wild. To this end, we measure

[‖] We use per-queue buffer occupancy as maximum queue length.

the buffer occupancy of 7 CCAs (the top-5 CCAs used in websites [193] plus two recent latency-sensitive CCAs, namely GCC and Copa), over real-world bandwidth traces [188]. We further measure the network RTT at the sender, and the application-layer performance (including the delay in the socket buffer and retransmissions). A lower RTT and application delay indicate that such a given CCA is more latency-sensitive. As we can see in Fig. 8.6, GCC, Copa, and Vegas have a low network RTT and application delay. Thus, delay-sensitive applications can choose these CCAs to achieve lower latency. Cubic, Yeah, and Illinois have a much higher delay, while BBR is in-between. We observe that the CCAs concentrated in three clusters (dashed circles in Fig. 8.6). Concretely, GCC, Copa, and Vegas have a queue occupancy of less than 20%; Cubic, Illinois, and Yeah have a queue occupancy of more than 80%; and BBR's queue occupancy stays in-between. Therefore, we set three queues and use the average queue occupancy in these three clusters as our targets $\{q_0^{(i)}\}$. We expect other CCAs to fall into one of these three representative categories, if not we can configure Confucius to work with more queues.

Practical challenges. While one can characterize flows offline as we did above, Confucius cannot use the same approach online. Indeed, Confucius works at line-rate and flows will not come prelabeled with their CCA. Inferring the buffer aggressiveness of a flow is challenging in practice for the following reasons. First, the buffer aggressiveness of flow may take a long time to manifest. For example, Confucius will not be able to characterize short flows lasting only a few RTTs (§8.2). Second, the network conditions will also affect the measurement, effectively deceiving Confucius. For example, a drop in the available bandwidth will result in an increase in the buffer occupancy [188], which does not necessarily mean that flow is aggressive in occupying the buffer. Finally, a flow's buffer aggressiveness

can change over time. For example, a Cubic flow throttled/congested elsewhere (on a different router) will not be aggressive in buffer occupancy (although Cubic would). Such a cubic flow can share the queue with other delay-sensitive flows. However, when the bottleneck moves to the current router, this Cubic flow will be aggressive on the buffer occupancy. Therefore, we need to periodically monitor the buffer share that each flow occupies within its current queue and re-consider its classification. We elaborate on our algorithm in the next subsection.

8.5.2 Hysteresis-based Adjustment

To allow re-classifications while avoiding oscillations in flows' classification, we introduce a hysteresis mechanism. The overall classification steps are as follows:

Classification of new flows. For the flow f in the new-flow queue $\mathcal{F}_{new}$, when the flow is ready (its weight reaching one) to be moved out from the new-flow queue Q_{new} to one of the old queues (which we elaborate on in §8.4.1), we measure the buffer occupancy of that flow q_f i.e. the number of packets of this queue that belong to flow f. We then find the queue i with the nearest Q_i to accommodate this flow.

Periodic adaptation. Confucius periodically examines flows and queues and moves flows accordingly. First, intra-queue examination identifies and moves flows that are outstanding among flows in the current queue (e.g., a flow that is more aggressive compared to the other flows). Second, the queue-level examination checks if the length of a queue fits the queue's control target.

1. **Intra-queue examination.** Confucius examines the buffer each flow occupies and compares it with its fair share. Specifically, if the buffer occupancy of a flow ($\frac{q_f}{\sum_{g \in Q_i} q_g}$) is larger than its fair share ($\frac{1}{|Q_i|}$), i.e.:

$$\frac{q_f}{\sum_{f \in Q_i} q_f} \geqslant \frac{1}{|Q_i|} + \alpha \tag{8.9}$$

where $\alpha > 0$ is a hysteresis, that flow is too aggressive in the current queue. Confucius wll promote that flow from queue Q_i to Q_{i+1} to keep Q_i near its control target. Similarly, a flow with an outstandingly lower buffer occupancy that its fair share in the queue, i.e.:

$$\frac{q_f}{\sum_{f \in Q_i} q_f} \leqslant \frac{1}{|Q_i|} - \alpha \tag{8.10}$$

will be demoted from queue Q_i to Q_{i-1}. Here we set α to 10% based on our previous observations in Fig. 8.6. Our evaluation in §8.7 shows that the performance of Confucius is not sensitive to the workloads and CCAs.

2. **Queue-level examination.** Confucius verifies that the length of each queue is within the target. If the length of a queue exceeds a safe region between the control target of any of the two neighbor queues, Confucius moves all flows in the current queue to a higher or lower queue, as shown in Figure 8.7. This is needed because the intra-queue examination only focuses on cross-flow relative occupancy. Thus, it cannot identify instances in which flows in the current queue are comparably aggressive but more aggressive than the target of this queue. For example, assume that there are two Cubic flows that were previously classified to Q_1 (the least aggressive) due to being throttled elsewhere or measurement errors. When these Cubic flows start to

be aggressive in buffer occupancy, Confucius would need to move them to a different queue to protect latency-sensitive flows that may join.

While seemingly complex, these operations are well within the capabilities of Linux-based edge routers. In fact, we have implemented a complete prototype in §8.6.

8.6 Confucius IMPLEMENTATION

Implementing Confucius in Linux kernel has some challenges. We discuss them and our solutions below.

Order-preserving during reclassification. Flows can be moved to another class in the runtime. Thus, we need to ensure the order-preservation during the reclassification of Confucius of a certain flow. In response, we adopt a virtual class design in Confucius. During the enqueue process of new packets, we bind the `sk_buff` to each flow. During the dequeue process, we search for all flows that are bound to the determined class and dequeue the packet with the earliest enqueue time. In this way, when moving a flow to another class, we can just rebind the pointer of the flow from the previous class to the new class.

Reducing computational overhead. To implement Confucius in Linux kernel and optimize the execution overhead, we need to strictly optimize the computational overhead. Specifically, we have the following two implementations:

(i) Bit-shifting for exponential operations. Confucius reweights flows based on their ages with an exponential function, yet the floating number calculation in the kernel is expensive. Therefore, we quantize the weight of new flows with the unit of $\frac{1}{128}$. We follow the imple-

mentation of EWMA and use bit shifts for the exponential changes of the weights, i.e., left shifting the weight by one bit every $\frac{1}{\lambda}$ milliseconds.

(ii) Periodical reweighting and reclassification. The reweighting and reclassification do not necessarily need to happen for each packet. For the reweighting, as we discussed before, we only need to reweight for a certain flow every $\frac{1}{\lambda}$ milliseconds. When we set $\lambda = 0.004$, this means to reweight every 250 ms. For the reclassification, we should at least observe the results after moving one flow to a new class for a certain period to measure the queue utilization, which should at least be more than one RTT to fully observe the behavior of the sender in the new class. Therefore, we also reclassify the flows in a periodic way – we set the reclassification period to 100ms.

8.7 Evaluation

We first present our experimental setup (§8.7.1); then we evaluate Confucius by answering the following questions:

- How does Confucius navigate the fairness-volatility trade-off compared to baselines on real-world Web traces? Confucius protects an HRT flow from delay degradation when competing with loading 95% of websites with various CCAs. In contrast, with classless schemes such as FQ or FIFO, the percentage is less than 30% (§8.7.2).
- How sensitive is Confucius to changes in workload? We vary the size and number of flows and find that Confucius remains consistently performant (in terms of delay degradation for the HRT flow and PLT degradation for Web flows) always following our theoretical analysis (§8.7.3).

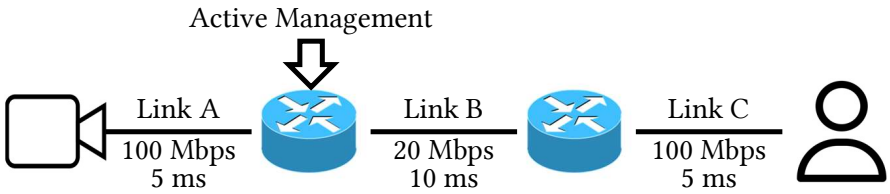
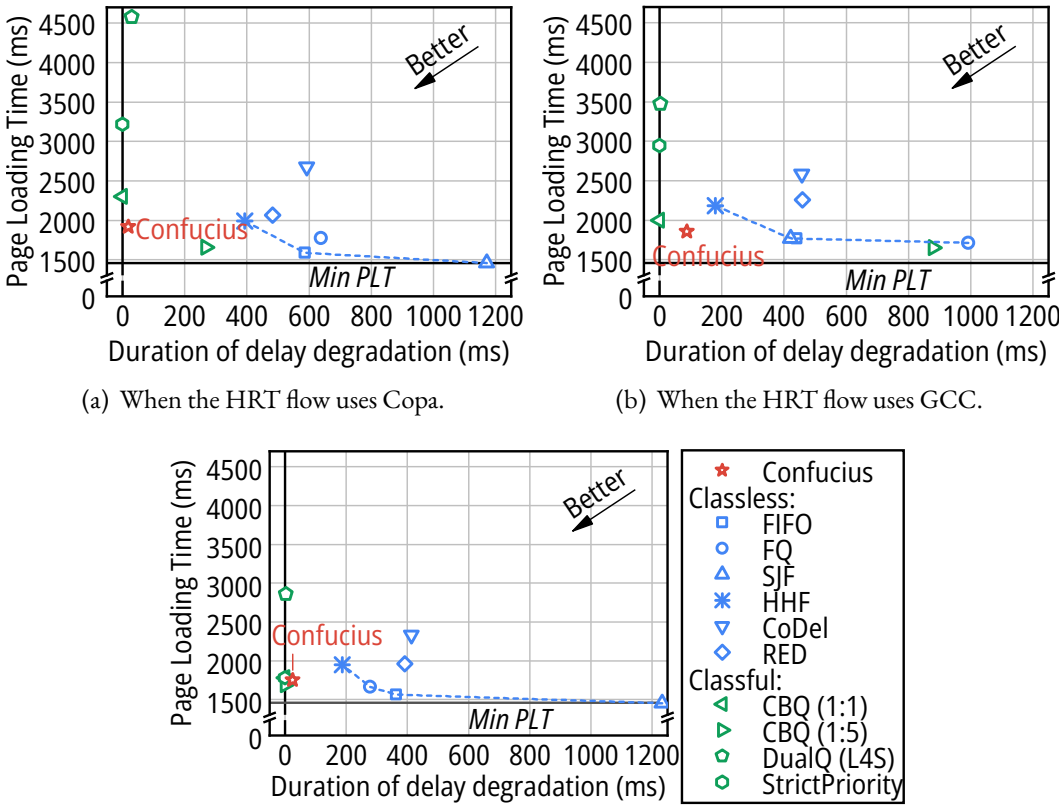


Figure 8.8: Experiment setup.

- How does Confucius scale if there are multiple flows with different CCAs? We test Confucius under the coexistence of flows with different CCAs, and demonstrate that Confucius can correctly separate flows based on their behaviors and provide consistent performance to all of them (§8.7.4).
- How does Confucius perform in the testbed prototype? We integrate Confucius into the `qdisc` module in Linux kernel 4.4.0 and evaluate Confucius with real HTTP request traces. Confucius can reduce the duration of delay degradation by more than 60% with reasonable overhead (§8.7.5).
- How does Confucius perform in different settings? We show that Confucius is still able to outperform baselines when working with multiple HRT flow competition, bandwidth-probing CCAs, and different bottlenecks (§8.7.6).

8.7.1 Experiment Setup

`Ns-3` **setup.** In §8.7.2-8.7.4, we evaluate the performance of Confucius with `ns-3.34`. We set up a linear topology and limit the capacity of the bottleneck link to 20Mbps, which is the average bandwidth in the WiFi traces from [188], as shown in Figure 8.8. The round-trip propagation delay is set to 40ms in total based on measurements from [188]. We further



(a) When the HRT flow uses Copa.

(b) When the HRT flow uses GCC.

(c) When the HRT flow uses BBR.

Figure 8.9: The trade-off between the performance of the HRT flow (duration of delay degradation) and Web flows (page loading time). The dashed line denotes the Pareto front of classless baselines. We change the CCA that the HRT flow uses in different subfigures and observe similar performance improvements of Confucius in all experiments.

change the RTT and the bottleneck in §8.7.6. We adopt a videoconferencing application in `ns-3`, of which the flow is an HRT flow. We connect the HRT flows to different delay-sensitive CCAs, including Copa [47], GCC [77], BBR [75] etc. The Web flows use the default CCA in Linux kernel – Cubic [129].

Linux kernel setup. In §8.7.5, we implement Confucius as a kernel module of queue disciplines (`qdisc`) in traffic control in Linux kernel 4.4.0 and evaluate the performance of Confucius on a machine with Intel Xeon E5-2620 v4 CPU. We run the official CCP-based implementation of Copa [46].

Web traces. To compose a realistic and relevant dataset of web traffic, we followed two steps. First, we collected the Alexa Top-1000 websites [34] (July 2022, distribution in Fig. 8.1). Second, we loaded each of these websites and measured the size of the HTTP requests they trigger. Having this dataset we replay the traces from these 1000 websites to test a variety of scenarios. We plan to release our dataset.

Baselines. We compare the performance of Confucius with multiple scheduling and AQM baselines. For the parameters in these baselines, we use the default parameters in the Linux kernel 4.4.0 or `ns-3.34`.

(1) `FIFO` and (2) `FQ`, the two most used schedulers.

(3) `SJF` (shortest job first) prioritizes short flows over long flows. Since we cannot know which job is shorter, we approximate a job's length with its age (namely, PIAS [51]), i.e. always prioritizing flows that are newer, which is exactly opposite to what Confucius tries to do.

(4) `HHF` [105] heavy-hitter filter differentiates between small flows and heavy-hitters, giving each category a fixed share of bandwidth.

(5) `CoDel` [203] and (6) `RED` [113] will drop packets before the queue overflows to notify the sender about the congestion.

(7) `CBQ` puts flows from different applications into different classes based on their labels. We set the weights for two classes to 1:1 and 1:5 and evaluate performance, respectively.

(8) `StrictPriority` strictly prioritizes traffic from HRT flows if they are labeled accordingly.

(9) `DualQ` [231] is a recently proposed scheduler in L4S [69] that protects latency-sensitive flows with labels.

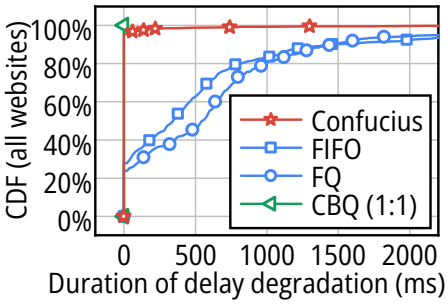
Metrics. We focus on the following metrics in experiments.

- **Duration of delay degradation** for video frames is the duration for which the delay of the video frame is greater than 190 ms. This directly reflects users' experiences on video stalls [188, 216, 290]. We use this metric to evaluate how volatility affects the performance of the HRT flow.
- **Page Load Time (PLT)** is the time till the last HTTP request in a web page is completed. We use this metric to evaluate the performance of web traffic. PLT degradation refers to the increase of delay compared to `FQ`.

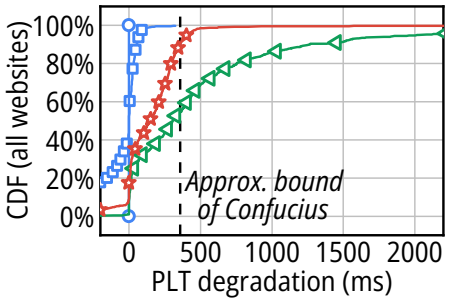
Besides, we also evaluate other metrics in different experiments, which we will elaborate on accordingly.

Simulation scenario. At t=0 we start an HRT flow from the videoconferencing application. At t=10s we reconstruct the requests associated with one of the Alexa Top websites. All flows are active i.e. we are not replaying pre-recorded traffic. We run the same scenario 1000 times, once per website. In each run, we measure the duration where the frame delay of the video flow is larger than 190 ms (delay degradation). We also measure the loading time of the web pages from different websites. We repeat the whole experiment three times, each considering a different CCA for the HRT flow. We summarize and present the average results in Figure 8.9.

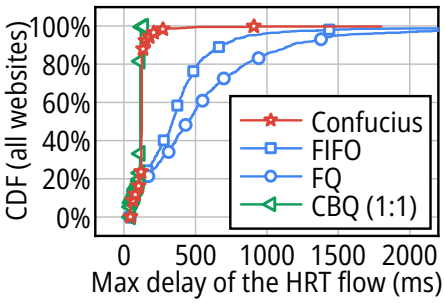
Confucius strikes a balance between video and web performance that is consistent across CCAs. In Figure 8.9(a), we observe that classless schedulers (i.e. those that do not use a label from the end host and are marked in blue) suffer from long video stalls. For example, when using `FQ` and `FIFO`, the video flow experiences delay degradation for 600 ms on average. Classful schedulers (i.e. those that require labels on packets and are marked in green) protect prelabeled video flows, but considerably degrade the PLT for the Web traffic. Worse yet, as we discussed in §8.2, it is unrealistic to assume that an end-host will correctly label all traffic. Confucius not only reduces the duration of stuttering compared to existing classless schemes, but is almost on par with classful schemes. Moreover, Confucius maintains a low PLT for Web flows. Notably, Confucius pushing the Pareto front of the classless schedulers (the dashed blue line) forward. The results are similar for Confucius when the video flow uses other CCAs such as BBR or GCC, as shown in Figures 8.9(b) and 8.9(c).



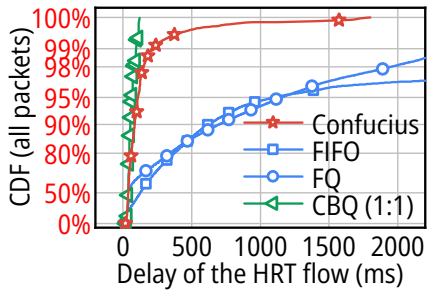
(a) Duration of delay degradation of the HRT flow (CDF over websites).



(b) PLT degradation of Web flows to FQ (CDF over websites).



(c) The max frame delay of the HRT flow when Web flows arrive (CDF over websites).



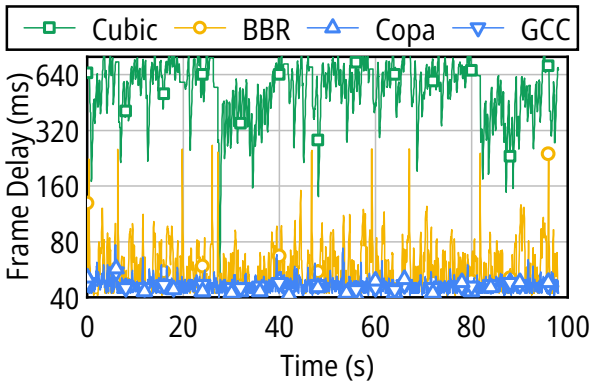
(d) The delay distribution during the period when Web flows arrive (CDF over packets, log-scaled).

Figure 8.10: The distribution of results in Fig. 8.9(a).

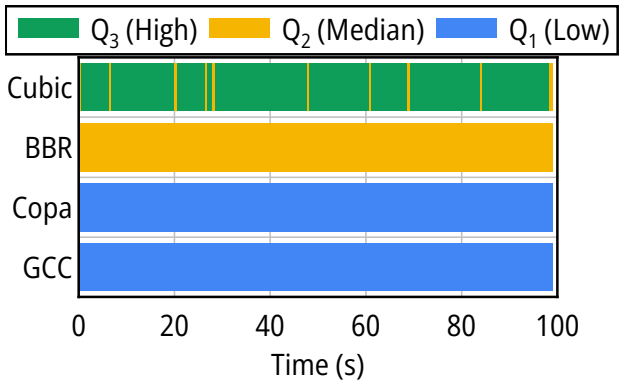
Confucius protects the HRT flow from traffic from more than 95% of the websites, while not sacrificing their performance. We further break down the distribution for different websites in Figure 8.9(a) into Figure 8.10. Figure 8.10(a) which presents the distribution of delay- degradation duration when the video flow encounters Web flows from different websites in the dataset. With FQ or FIFO, the HRT flow will experience delay degradation (frame delay >190ms) for more than 70% of websites, half of which will even last 520 ms (in the case of FIFO) and 660 ms (in the case of FQ). In contrast, with Confucius, the HRT flow will not experience any delay degradation when encountered with 95% of the websites, comparable to CBQ. Importantly, Confucius does not over-penalize web traffic – the PLT of 90% of websites are only increased by less than 360 ms against FQ, as shown in Figure 8.10(b), which mostly corroborate our previous theoretical analysis. We further present the distribution of maximum experienced delay, and the delay of all packets of the HRT flow when competing with different websites in Figures 8.10(c) and 8.10(d). This further demonstrates that Confucius is able to control the latency volatility in not only the duration of delay degradation but also directly the raw delay. The results when using GCC and BBR are similar.

8.7.3 Confucius UNDER WORKLOAD CHANGES

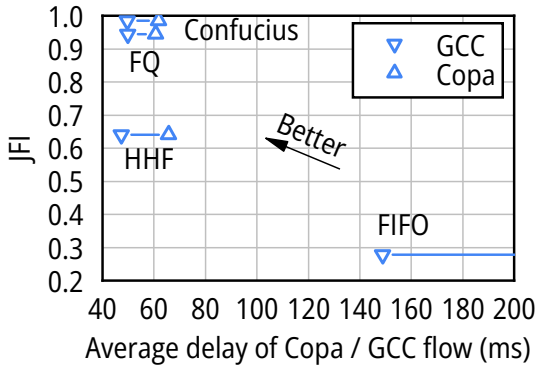
In this subsection, we test our theoretical analysis in a more practical setting. Concretely, we investigate whether Confucius can provide consistent performance in different workloads. To this end, we vary the workload by changing the number of flows in a Web page and the size of Web flows. We measure the duration of delay degradation in different scenarios and the degradation on the PLT against FQ.



(a) The frame delay of different flows.

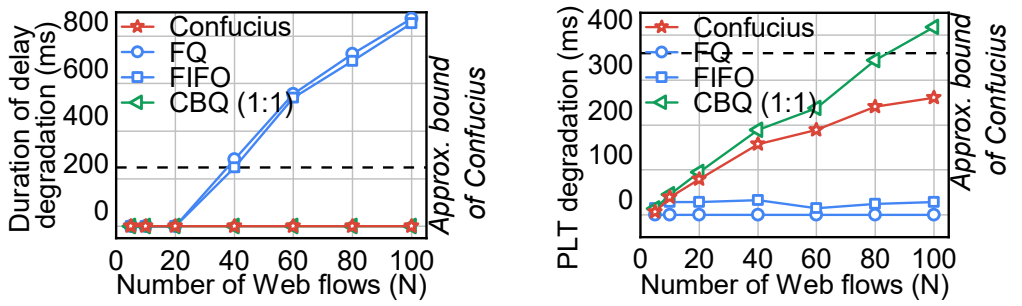


(b) The classification results in different time.



(c) The JFIs and delays among baselines.

Figure 8.11: Four flows with different CCAs (Cubic, BBR, Copa, and GCC) run in the same bottleneck router. We present the frame delay and classification results of these flows when using Confucius over time in Figure 8.11(a) and 8.11(b). We also compare the fairness (JFI) and the delay of latency-sensitive flows (Copa and GCC) of Confucius and baselines in Figure 8.11(c).



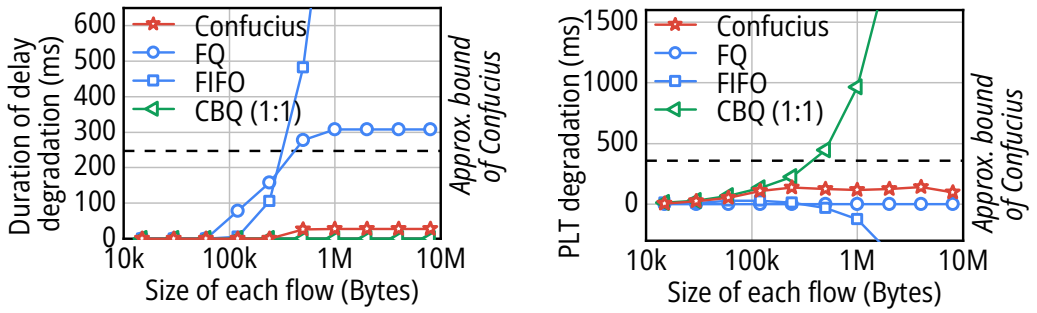
(a) Duration of delay degradation for the HRT (old) flow.

(b) PLT degradation of Web (new) flows against FQ.

Figure 8.12: Performance consistency in workloads with different *number* of Web flows, each flow with the size of 15KB.

Confucius delay degradation is bounded by a theoretically-estimated threshold, confirming our analysis. We vary the number of flows in the Web page from 5 to 100, each with the size of 15KB and summarize our results in Figure 8.12(a). The duration of delay degradation for FQ and FIFO increases with the number of flows. For example, when the number of Web flows goes to 60, the HRT flow experiences a degraded delay for more than half a second when using FQ or FIFO. On the contrary, Confucius maintains zero delay degradation in this setting, similar to CBQ (which uses labels). We further compare the experimental results with our previous analysis in §8.4.2. As we can see in the yellow dashed line in Figure 8.12, the experimental results corroborate our previous theoretical analysis on the performance of Confucius in Table 8.2.

We further change the size of Web flows (from short flows to long flows) and see if Confucius is capable of handling all types of competing traffic. We vary the size of Web flows from 15KB to 9MB, and run 5 flows with the same size to compete with the HRT flow. With the increase of the size of flows, the competing flows are changing from short flows



(a) Duration of delay degradation of the HRT (old) flow.

(b) PLT degradation of Web (new) flows against FQ.

Figure 8.13: Performance consistency in workloads with different *size* of Web flows, each experiment having 5 flows.

(e.g., Web) to long flows (e.g., FTP). In this case, when using FIFO, the HRT flow will suffer from drastic delay degradation due to failure to provide inter-CCA fairness across flows, as shown in Fig. 8.13(a). The HRT flow using FQ also has a long delay degradation of hundreds of milliseconds. In contrast, Confucius is still able to achieve both negligible duration of delay degradation for the HRT flow and bounded degradation of the PLT for the Web flows in the same time.

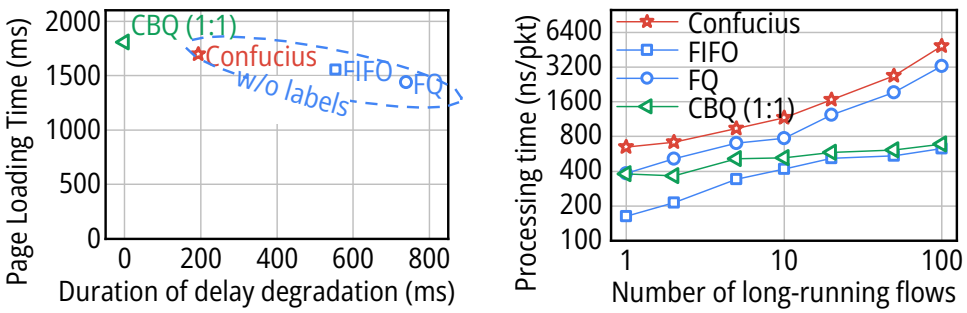
8.7.4 Heterogeneous Flow Classification

In this subsection, we zoom in on Confucius's flow classification mechanism and investigate its effect on delay and fairness. We find that Confucius groups flows of the same CCA together, without any prior knowledge, which in turn leads to better performance compared to the baselines.

We simultaneously run HRT flows of four different CCAs: one Cubic flow, one BBR flow, one GCC flow, and one Copa flow for 100 seconds. We plot the frame delay for

each flow over time in Figure 8.11(a). In this experiment, we also measure the JFI in Figure 8.11(c) to present the fairness when using different schemes. We also compare the results (the delay of the Copa and GCC flow, and the JFI among all flows) of the same experimental settings with other schedulers in Figure 8.11(c). We find that with Confucius the Copa and GCC flows maintain a low end-to-end delay even though they share the bottleneck link with Cubic and BBR. Meanwhile, they also enjoy a reasonable fair share of the bandwidth – the JFI in this experiment is 0.98 in Figure 8.11(c).

To understand Confucius's superior performance, we look at its classifications over time and verify that Confucius works in practice as we expect. We make two observations. First, Confucius can classify flows using different CCAs into different queues. As shown in Figure 8.11(b), the Copa and GCC flows can be stably classified into the low occupancy queue (Q_1, blue), the BBR flow into the median occupancy queue (Q_2, yellow), and the Cubic flow into the high occupancy queue (Q_3, green). This follows our previous observation in Figure 8.6 – Copa and GCC both demonstrate similar low buffer occupancy, while Cubic occupies the buffer aggressively, and BBR in the middle. In this way, flows with different queue occupancy can be isolated from each other. Moreover, we notice that the Cubic flow can temporarily be in the same queue as BBR, as shown in the yellow lines in the green bar in Figure 8.11(b). This is, in fact, beneficial for Confucius as the Cubic flow has (at times) a low queue occupancy in its probing period. Second, flows with different CCAs can coexist in the same queue as long as they have similar buffer occupancy. In this experiment, Copa and GCC flows are put into the same queue since they have similar buffer occupancy. As we can see in Figure 8.11(a), these two flows still have consistent low latency all the time.



(a) HRT flow's delay degradation vs. Web flows' load time.

(b) Processing time for each packet. Axes are log-scaled.

Figure 8.14: Results over our Linux kernel-based testbed.

8.7.5 Testbed Experiments

We also evaluate the performance of Confucius in the Linux kernel. We find that Confucius is capable of achieving significant benefits in kernel-based implementations while only adding marginal processing delay.

We run an iperf3 flow, set the CCA to Copa, and measure the delay reported by iperf3 for the latency-sensitive flow. We then set up an HTTP server based on Python to serve the client with the Web traces we collected. We also measure the computational overhead of Confucius and the baselines. We log the processing time for the enqueue and dequeue operation in Linux `tc`, where the reweight and reclassification in Confucius are both implemented.

As shown in Figure 8.14(a), Confucius reduces the duration of delay degradation by more than 60% without the need for labels on each packet. This result is similar to our simulation in Figure 8.9(a). Moreover, 86% of websites when using Confucius do not suffer from delay degradation. Notably, this number is only 56% and 30% for FIFO and FQ.

We vary the number of long-running flows to observe how the processing time changes. Note that the processing time of Confucius is insensitive to the number of short flows, as they all belong to the new-flow queue. As shown in Figure 8.14(b), Confucius slightly increases the processing time for each packet compared to FQ. However, even if there are 100 concurrent long-running flows on the same queue discipline, the per-packet processing time is still 5 μs, indicating a processing rate of 200 kpps, or a bitrate of 100 Mbps$\sim$2.4 Gbps (depending on the packet size). Note that Confucius is mainly designed to be deployed on the last-mile routers such as home routers. This can satisfy the daily usage of home access points or last-mile routers. We stress that the kernel implementation of Confucius can be further optimized for high-performance execution in the future. We leave the further exploration of Confucius over numerous flows (e.g., in the routers in the core network) in the future.

8.7.6 Microbenchmarks

We further evaluate the performance of Confucius in a series of microbenchmarking settings. In Appx. E.2.1, we demonstrate that the hysteresis mechanism of Confucius (§8.5.2) is able to work with bandwidth-probing CCAs (e.g., BBR) and stably and correctly classify flows. We further show that Confucius will not have any side effects if the bottleneck is not the router where Confucius is deployed in Appx. E.2.2. Finally, we also show that even if there are multiple HRT flows competing at the same time, Confucius is still able to handle those flows simultaneously and provide significant performance improvements against baselines (Appx. E.2.3).

8.8 Summary

In this chapter, we propose Confucius, the first queue management scheme to balance fairness against volatility. Confucius achieves this by grouping flows based on their latency preferences, which it infers by observing their buffer occupancy over time. Confucius gradually adjusts per-flow weight, and uses those weights to devise the per-queue service rate. Doing so allows Confucius to mitigate volatility that degrades the performance of HRT flows. Linux kernel-based emulation and ns-3 based simulations show that Confucius can reduce the number of websites causing delay degradation for video flows from 70% to 5% with negligible overhead.

9

Conclusions and Future Work

9.1 Work Summary

Real-time multimedia transmission is an important application on the current Internet. As people's demand for a better life increases, the latency requirements for real-time multimedia transmission applications are becoming higher and higher. Optimizing latency for real-time multimedia transmission is of great significance. Existing solutions mostly focus on

median, 90th percentile, and other general latency cases, while neglecting the optimization of 99.9th percentile and even 99.99th percentile latency. In many real-time multimedia applications such as cloud gaming, remote surgery, and virtual reality, a one-in-ten-thousand stutter can have serious consequences. In comparison, the real-time multimedia transmission latency optimization in this work focuses more on stutter events with occurrence frequencies of one in a thousand and one in ten thousand.

Unlike median and 90th percentile latency, which have clear bottlenecks (usually propagation delay), when discussing extreme tail latency with occurrence probabilities of one in a thousand and one in ten thousand, any high latency component can lead to an increase in extreme tail latency, resulting in a decrease in user experience. This paper first analyzes the components and roles of existing real-time multimedia transmission in the Internet architecture and proposes the importance of controlling path latency. In a network with continuous fluctuations, decisions need to be made frequently at the end. If the endpoint's decision is slower due to high control path latency, it will lead to an increase in extreme tail latency. This paper divides the control path into feedback and decision components and optimizes them separately. This paper also emphasizes the difficulty of meeting extreme tail latency requirements with existing data path architectures and locates and optimizes the causes of latency fluctuations in the application layer, transport layer, and network layer.

The main research content and contributions of this paper can be summarized as follows:

1. In the control path, Zhuge, a congestion signal early feedback solution that shortens the feedback loop, is proposed to address end-to-end latency fluctuations caused by feedback latency fluctuations. Zhuge decouples the feedback loop from the data path

to achieve the goal of shortening the feedback loop for early congestion signal feedback. Specifically, this work classifies real-time multimedia transmission protocols based on their feedback modes into in-band and out-of-band feedback, and optimizes different types of feedback modes accordingly. Experiments based on real routers and large-scale simulations show that the early congestion signal feedback solution proposed in Chapter 4 can effectively reduce end-to-end latency fluctuations, thus improving user experience. This work was published at the ACM SIGCOMM 2022 conference and was tested for product deployment at Alibaba, achieving good performance improvements.

2. In the control path, Metis, a lightweight and reliable rate control decision framework, is proposed to address end-to-end latency fluctuations caused by decision latency and instability. Metis is a lightweight and reliable rate control decision conversion and interpretation framework that transforms optimized complex rate control decision algorithms into simple rate control decision algorithms, achieving timeliness and reliability of decision-making. Specifically, this work converts existing complex rate control decision algorithms based on machine learning and integer programming into simple rate control decision algorithms based on decision trees. Experiments and analysis based on existing algorithms show that the lightweight and reliable rate control decision conversion and interpretation framework proposed in Chapter 5 can effectively reduce performance fluctuations, thus improving user experience. This work was published at the ACM SIGCOMM 2020 conference and was tested and deployed in real production environments at Tencent, Kuaishou, and other companies.

3. In the data path, AFR, an adaptive frame rate adjustment solution, is proposed to address end-to-end latency fluctuations caused by latency fluctuations in the application layer's video codec. AFR is an adaptive frame rate adjustment solution that actively adjusts the frame rate of the video codec in the application layer, thereby reducing latency fluctuations in the video codec. Specifically, this work proposes an application-layer active queue management solution based on joint analysis of network conditions and application conditions, using queuing theory and stochastic process modeling. Experiments for large-scale users show that the adaptive frame rate adjustment solution proposed in Chapter 6 can effectively reduce end-to-end latency fluctuations in cloud gaming applications. This work was published at the USENIX NSDI 2023 conference and has been deployed at Tencent on a large scale for two years.

4. In the data path, Hairpin, a joint recovery solution that integrates multiple packet loss recovery mechanisms, is proposed to address end-to-end latency fluctuations caused by transport layer packet loss and its recovery mechanisms. Hairpin is a joint packet loss recovery solution that integrates existing packet loss recovery mechanisms, especially retransmission and redundancy recovery. Specifically, this work uses Markov chains to jointly model packet loss and retransmission, proposing an optimal strategy for adding redundancy and deciding whether to retransmit. Experiments based on real network datasets show that the joint packet loss recovery solution proposed in Chapter 7 can effectively reduce end-to-end latency fluctuations while also reducing the cost of bandwidth overhead. This work is accepted by USENIX NSDI 2024 conference.

5. In the data path, Confucius, a new router queue management solution, is proposed to address end-to-end latency fluctuations caused by bursty competition and queuing of multiple applications at the network layer. Confucius is a new router queue management solution that reduces end-to-end latency fluctuations by optimizing differential service bandwidth allocation without relying on endpoint information. Specifically, this work infers the latency sensitivity of different flows by observing the bottleneck queue occupancy of different flows, thereby achieving differential service optimization for different flows. Tests based on real routers and thousands of websites show that the router queue management solution proposed in Chapter 8 can effectively reduce end-to-end latency fluctuations without relying on any endpoint labels or information.

9.2 Future Work

Real-time multimedia is a long-standing research topic in network systems, but the application scenarios it faces are becoming more and more complex. From network telephony to video conferencing, to cloud gaming, remote surgery, and finally to virtual reality and augmented reality, the latency requirements of applications for networks are getting higher and higher, and the scenarios are becoming more diverse. Real-time multimedia applications involve a series of deep systemic issues, some of which are not just research problems in the field of networks. This paper only solves some key problems, but there are some aspects that can be further explored in the future.

1. Joint optimization with the operating system. With the gradual promotion of edge node deployment and the large-scale deployment of new-generation wireless access net-

work technologies (such as WiFi 6 and 5G), the net propagation delay of networks is getting lower and lower. At this point, the latency bottleneck on the endpoint side becomes more prominent. Of course, many excellent researchers in the field of operating systems are also trying to reduce this latency, but it should be noted that network latency is actually the most elastic part of latency components: the network can always sacrifice some throughput for lower latency. Therefore, if the latency budget of the entire link can be planned in advance when the endpoint operating system and other latency bottlenecks are anticipated, latency can be further reduced.

2. Joint optimization with different scenarios. Network layer indicators are currently more related to network service quality. Even if the stutter rate and other indicators are actually counted at the application layer video frame granularity, this is not the user's real experience, but only an estimate of the user experience. Furthermore, different users may have different experiences with the same latency and picture quality due to differences in their physiological and psychological states and application usage. How to understand the user's real experience and optimize it, especially when these emerging application scenarios are gradually entering people's field of vision, is also a direction worth further in-depth research.

From a broader perspective, the latency problem solved in this paper is not only applicable to real-time multimedia transmission. In fact, the design of transport layer, network layer, and control path in this work can be migrated to other applications with similar low-latency requirements. In recent years, new network scenarios such as the Internet of Things and connected vehicles have brought great opportunities to network research. Whether the low-latency optimization in this work can be applied to other network scenarios and

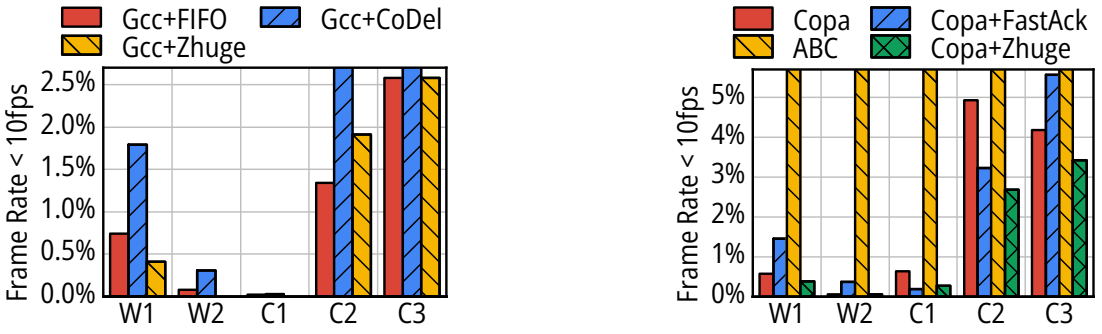
whether there are new challenges to be solved are also directions worth exploring in the future.

A

Zhuge (§4)

A.1 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 §4.2.3, and the trace collection of available bandwidth from WiFi networks in §4.7.2. We present their measurement details as below.



(a) RTP/RTCP. (b) TCP.

Figure A.1: The ratio of frame rate<10fps over real-world traces.

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 §4.2.3.

Available bandwidth of WiFi networks. We measure the available bandwidth of the WiFi network in a nearby restaurant [28], 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.

A.2 Supplementary Trace-Driven Simulations

Frame-rate improvements. We further present the summary of the performance improvements on the frame-rate in Figure A.1. We measure the ratio of low frame-rate (per-second

	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 A.1: Performance of on the original traces of ABC.

frame rate $<$10fps). As shown in Figure A.1(a) and A.1(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.

Results over the traces used in ABC [125]. 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 §4.7.2. Thus, the traces in ABC may not faithfully reflect the development of the wireless access networks in recent years.

B

Metis (§5)

B.1 Resampling in Decision Tree Training

To explain the resampling equation during decision tree training (Equation 5.1), we first briefly introduce the basic knowledge about RL used in this paper. We refer the readers to [248] for a more comprehensive understanding of RL.

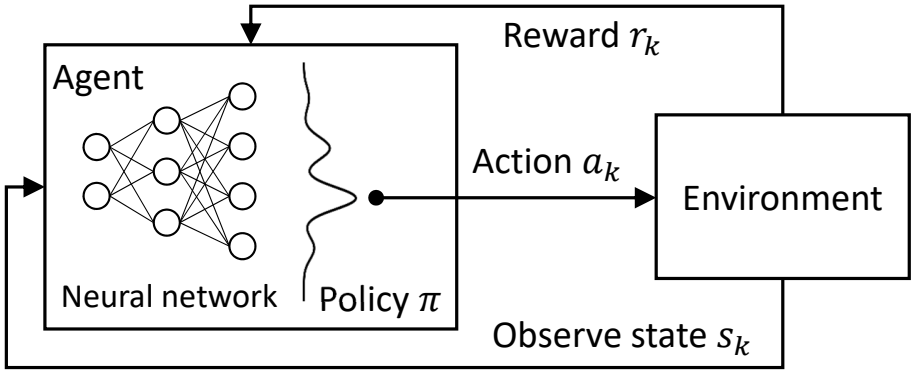


Figure B.1: RL with neural networks as policy.

In RL, at each iteration t, the *agent* (e.g., a flow scheduler [83]) first observers a *state* $s_t \in \mathcal{S}$ (e.g., remaining flow sizes) from the surrounding *environment*. The agent then takes an *action* $a_t \in \mathcal{A}$ (e.g., scheduling a flow to a certain port) according to its *policy* π (e.g., shortest flow first). The environment then returns a *reward* r_t (e.g., FCTs of finished flows) and updates its state to s_{t+1}. Reward is used to indicate how good is the current decision. The goal is to learn a policy π to optimize *accumulated future discounted reward* $\mathbb{E}\left[\sum_t \gamma^t r_t\right]$ with the discounting factor $\gamma \in (0, 1]$. $\pi_\theta(s, a)$ is the probability of taking action a at state s with policy π_θ parameterized by θ, which is usually represented with DNNs to solve large-scale practical problems [195, 196]. An illustration of RL is presented in Figure B.1.

However, it is not easy for the agent to find out the actual reward of a state or an action in the training process since the reward is usually *delayed*. Therefore, we need to estimate the potential *value* of a state. *Value function* $V_t^{(\pi)}(s)$ is introduced to determine the potential future reward of a state s at the time t with the policy π:

$$V^{(\pi)}(s) = R(s) + \sum_{s' \in \mathcal{S}} p\left(s' | s, \pi(s)\right) V^{(\pi)}(s') \tag{B.1}$$

where $p(s'|s,a)$ is the transition probability onto state s' given state s and subsequent action a. Similarly, *Q-function* $Q_t^{(\pi)}(s,a)$ is to estimate the value of how a certain action a at state s may contribute to the future reward:

$$Q^{(\pi)}(s,a) = R(s) + \sum_{s' \in \mathcal{S}} p\left(s'|s,a\right) V^{(\pi)}(s') \tag{B.2}$$

Therefore, a good action a at the state s would maximize the difference between the value function and Q-function, i.e., the optimization loss $\ell(s,\pi)$ of RL could be written as:

$$\ell(s,\pi) = V^{(\pi)}(s) - Q^{(\pi)}(s,a) \tag{B.3}$$

In the teacher-student learning optimization in §5.3.2, to make the loss independent of π and therefore easy to optimize, Bastani et al. [58] bounded the loss above with:

$$\tilde{\ell}(s) = V^{(\pi)}(s) - \min_{a' \in A} Q^{(\pi)}(s,a') \geqslant V^{(\pi)}(s) - Q^{(\pi)}(s,a) \tag{B.4}$$

Therefore, we can resample the (state, action) pairs with the loss function above, which explains the sampling probability in Equation 5.1. The sampling probability $p(s,a)$ in Equation 5.1 is proportional to but not equal to the loss function due to the normalization of probability.

We further empirically evaluate the improvement on QoE of the resampling step. We measure the QoE of the decision trees with and without the resampling step. As shown in Figure B.2, 73% of traces could benefit from the resampling step with different degrees of improvement. The median improvement on QoE over all traces is 1.5%. Since the resam-

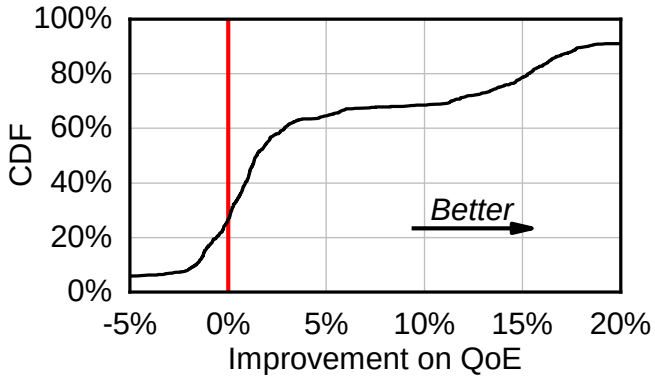


Figure B.2: The resampling step could improve the QoE of 73% of the traces, with the median improvement of 1.5%.

pling step is adopted for the last mile performance improvement, network operators may choose to skip the step if performance is not a critical issue for them.

B.2 Implementation Details

Parameter settings. For the DNN in Pensieve, we set the number of leaf nodes (M) to 200. Our experiments on the sensitivity of M in Appendix B.5 shows that a wide range of M perform well.

Testbed details. We train the decision tree with `sklearn` [211] and modify it to support the CCP. The server for Pensieve is equipped with an Intel Core i7-8700 CPU (6 physical cores) and an Nvidia Titan Xp GPU.

B.3 Pensieve Debugging Deep Dive

We also provide more details on the experiments of two links with bandwidth fixed to 3000kbps and 1300kbps in §5.5.4.

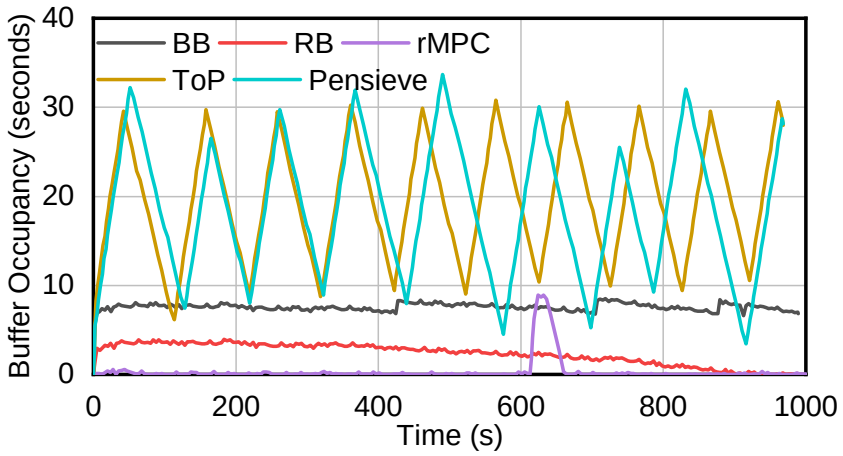


Figure B.3: Buffer Occupancy at 3000kbps Link.

3000kbps link. Except for the experiments in §5.5.4, we also investigate the runtime buffer occupancy over the 3000kbps link. As shown in Figure B.3, the buffer occupancy of Pensieve fluctuates: buffer increases when 1850kbps is selected and decreases when 4300kbps is selected, which is also faithfully mimicked by Metis +Pensieve. The oscillation leads to a drastic smoothness penalty. Meanwhile, the buffer occupancy can also interpret the poor performance of rMPC in Figure 5.10: rMPC converges at the beginning. Thus, there is no enough buffer against the fluctuation of chunk size since the size of each video chunk is not the same. Thus a substantial rebuffer penalty is imposed on rMPC. The buffer of BB and RB decreases slightly during the total 1000 seconds experiment as the goodput is not exactly 2850kbps (the average bitrate of sample video).

As the raw outputs of the DNNs in Pensieve are the normalized probabilities of selecting each action, we further investigate those probabilities of Pensieve on the 3000kbps link and present the results in Figure B.4. A higher probability close to 1 indicates higher confidence in the decision. We can see that Pensieve does not have enough confidence in the decision

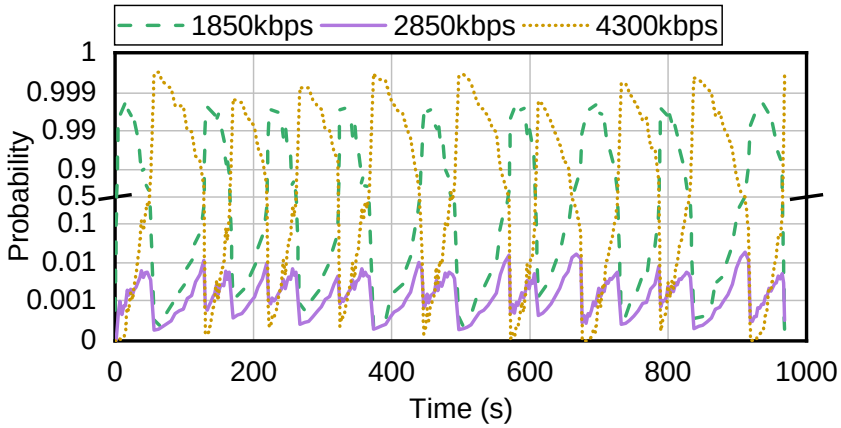


Figure B.4: Probabilities of selecting 1850kbps, 2850kbps, 4300kbps qualities. The probability of selecting other three qualities is less than 10^{-4} thus not presented.

it made, which suggests that Pensieve might not experience similar conditions in training; thus, it does not know how to make a decision.

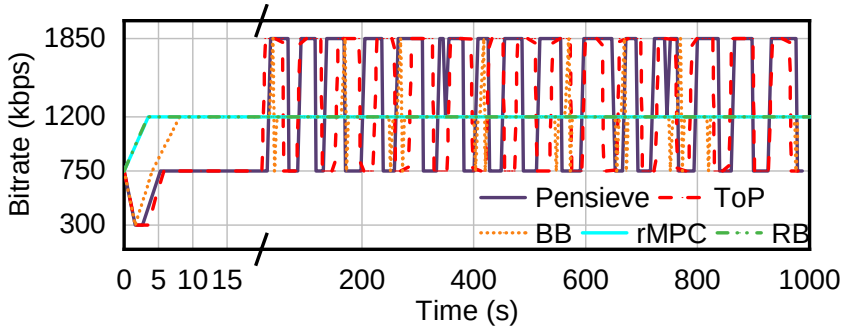
BB	RB	rMPC	Metis +Pensieve	Pensieve
1.050	0.904	0.803	0.986	0.983

Table B.1: QoE on the 1300kbps link.

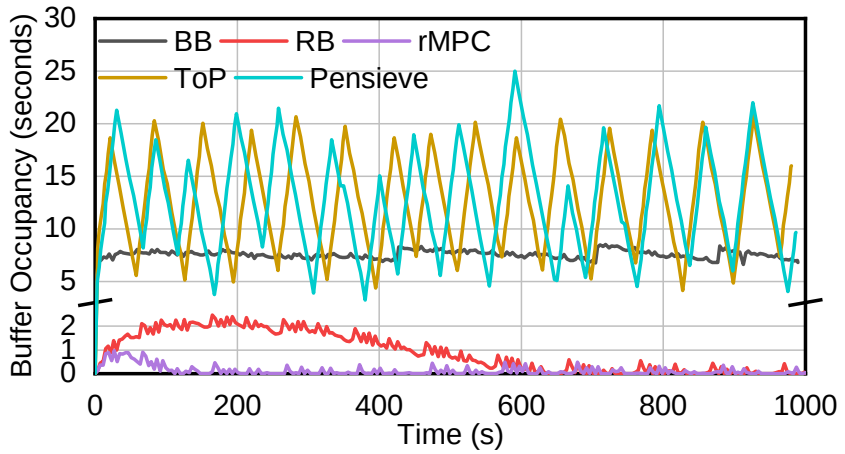
1300kbps link. We also provide the details about the experiments in Figure 5.9(c) on a 1300kbps link and present the results in Figure B.5 and Table B.1. The results are similar to the 3000kbps experiment, except that the performance of RB is worse since it converges faster.

B.4 Interpretation Baseline Comparison

We further want to know the reason for the performance maintenance of Metis. We measure the accuracy and root-mean-square error (RMSE) of the decisions made by Metis com-



(a) Bitrate

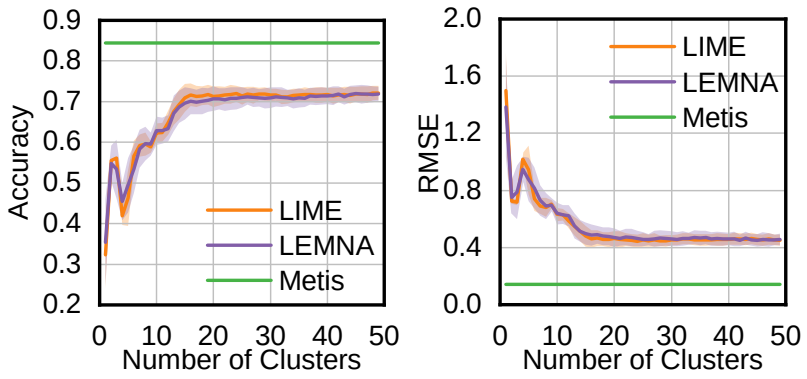


(b) Buffer Occupancy

Figure B.5: Results on a 1300kbps link. Better viewed in color.

pared to the original decisions made by DNNs. As baselines, we compare the faithfulness of Metis over the DNNs with two recent interpretation methods:

- LIME [217] is one of the most widely used blackbox interpretation method in the machine learning community. LIME interprets the blackbox model with the linear regression of the inputs and outputs.
- LEMNA [128] is an interpretation method proposed in 2018 and designed to interpret DL models based on time-series inputs (e.g., RNN). LEMNA employs a



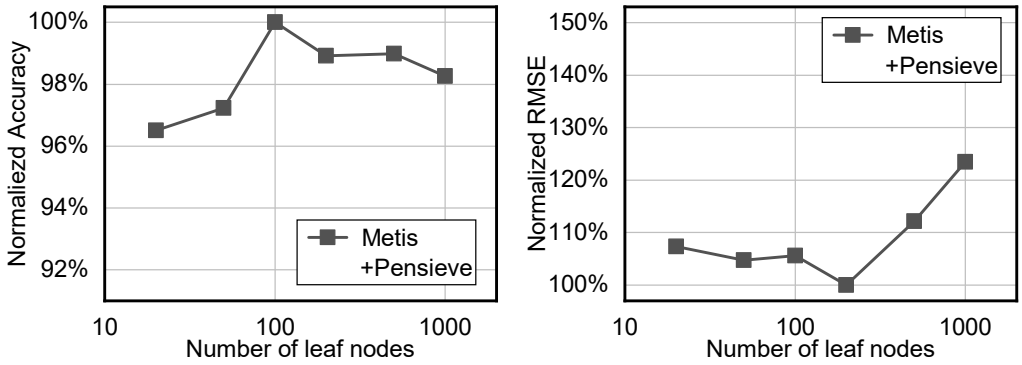
(a) Accuracy (Pensieve). (b) RMSE (Pensieve).

Figure B.6: Faithfulness of Metis. Shaded area spans $\pm$ std. Higher accuracy and lower RMSE indicate a better performance. Better viewed in color.

mixture regression to handle the dependency between inputs. We employ LEMNA as a baseline since some networking systems also handle time-series inputs.

As both methods are designed based on regressions around a certain sample, to make a fair comparison, we run the baselines in the following way: At the training stage, we first use k-means clustering [177] to cluster the input-output samples of the DL-based networking system into k groups. We then interpret the results inside each group with LIME and LEMNA. We vary k from 1 to 50 and repeat the experiments for 100 times to eliminate the randomness during training. Results are shown in Figure B.6. Since the decision tree interpretations of Metis do not rely on a particular sample, they do not need to be clustered and are constant lines.

From Figures B.6(a), Metis +Pensieve achieve high accuracy of 84.3% compared to original DNNs. The low decision errors in Figures B.6(b) indicate that even for those decision tree decisions that are different from DNNs, the error made by Metis is acceptable, which will not lead to drastic performance degradation. The accurate imitation of original DNNs



(a) Normalized Accuracy. (b) Normalized RMSE.

Figure B.7: Sensitivity of leaf nodes on prediction accuracy and RMSE. Results are normalized by the best value on each curve.

with decision trees results in the negligible application-level performance loss in §5.5.5. Meanwhile, the accuracy and RMSE of Metis are much better than those of LIME and LEMNA. Our design choice in §5.3.1 is thus verified: decision trees can provide richer expressiveness and are suitable for networking systems.

B.5 Sensitivity Analysis

In this section, we present the sensitivity analysis results on the hyperparameters of Metis when applied to the DL-based networking systems.

To test the robustness of Metis against the number of leaf nodes, we vary the number of leaf nodes from 20 to 5000 and measure the accuracy and RMSE for Pensieve. The results are presented in Figure B.7. The accuracy and RMSE of Metis +Pensieve with the number of leaf nodes varying from 20 to 5000 are better than the best results of LIME and LEMNA in Figure B.6 in Appendix B.4. The robustness indicates that network operators

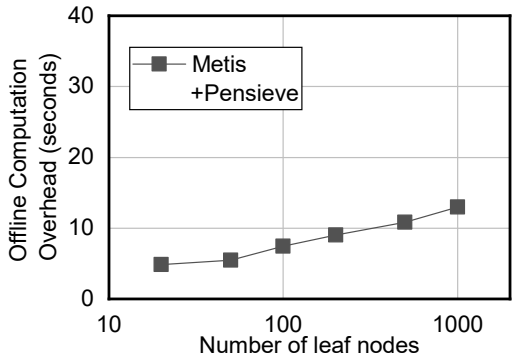


Figure B.8: Offline Computation Overhead of Metis with different number of leaf nodes.

do not need to spend a lot of time in finetuning the hyper-parameter: a wide range of settings all perform well.

B.6 Computation Overhead

We further examine the computation overhead of Metis in decision tree extraction. We measure the decision tree computation time of Pensieve at different numbers of leaf nodes on our testbed. As the action space of Pensieve (6 actions) is much small, the decision tree of Metis +Pensieve has completely been separated with around 1000 leaf nodes. Thus we cannot generate decision trees for Metis +Pensieve with more leaf nodes without enlarging the training set. As shown in Figure B.8, even when we set the number of leaf nodes to 5000, the computation time is still less than one minute. Since decision tree extraction is executed offline after DNN training, the additional time is negligible compared to the training time of DNN models (e.g., at least 4 hours in Pensieve with 16 parallel agents [179]). Metis can convert the DNNs into decision trees with negligible computation overhead.

C

AFR (§6)

C.1 Potential Solutions and Concerns

In this section, we discuss why other potential solutions are insufficient to address the problem in this paper, and discuss other concerns of adapting the frame rate during runtime.

C.1.1 Potential Solutions

Discarding frames or adjusting resolutions. For most widely adopted codecs, dropping one frame or changing the resolution will make the following frames fail to recover the raw pixels of the block because they are differentially encoded by the motion vector to the previous one*. This is to utilize the redundant information between frames to reduce the bitrate of the stream. Since key frames do not rely on previous frames, they are usually much larger than other predictive frames (sometimes 10×) [158]. Therefore, given the same bottleneck bandwidth, sending a frame with 10× larger size will take approximately 10× time (tens to hundreds of milliseconds), which drastically increases the delay for the users. Moreover, frequently requesting key frames will degrade the goodput of the streaming and potentially increase the congestion in the network. Therefore, directly dropping delayed frames at the client or frequently changing the resolution will introduce stalls for the subsequent frames and degrade the users' experiences of high-quality RTC.

Adjusting the bit-rate. Without changing the resolution and frame rate, adjusting the bit rate has a very limited effect in reducing the decoding delay. Generally speaking, resolution, bit rate, and frame rate could be independently set. The display resolution describes the number of distinct pixels in each dimension that can be displayed, and the frame rate represents the number of pictures within one second of video. And the bit rate represents the amount of data used for storing the coded bit-stream. So the higher resolution we set, the more pixels a single picture will have, which could mean a higher definition of the video. And setting a higher frame rate means there will be more pictures per video second to make

*Recent advances on scalable video coding could partially break the inter-frame dependency, yet degrades video quality with the same bit-rate [234].

the video smoother. If we set a higher target bit-rate while keeping other parameters unchanged, the encoder can use more data to represent the pictures to achieve lower possible image distortion with a lower *quantization parameter* [49].

In this case, with the unchanged frame rate and resolution, the decoding procedure is also unaffected. For example, in H.264/AVC, a sequence of macroblocks can be composed of a slice, a picture, therefore, is a collection of one or more slices. Slices are completely independent of each other, and the macroblocks inside a video frame can be reconstructed in parallel. The video decoding has been parallelized using slice-level or block-level parallelism. The resolution will affect how many pixels there are in one frame, and the frame rate determines the tolerable decoding delay for each frame. The parallelized decoder is not significantly affected by the precision of each pixel. We further measure the decoding performance with different bitrates in production in Appendix C.2.4.

Preset the frame rate and resolution based on client types. An alternative to AFR is that the application checks whether the hardware could reliably decode the video at a certain resolution and frame rate at initialization. This, however, would lead to underutilization on the client side. The decoding capability of hardware is fluctuating over time due to various reasons. For example, we measure the distribution of decoding delay of each user session in Appendix C.2.5. One-fourth of users will have at least 1‰ time of a long decoding delay of $>$18ms, which could result in severe queuing delay, as we illustrated in Figure 6.6. In this case, if we set the resolution and frame rate based on this tail metric, users will have a much lower resolution and frame rate during most of the time. Therefore, we need to control the frame rate in the runtime to dynamically adapt to the network and decoder dynamics.

Allocating the application with dedicated resources. Another seemingly feasible solution is to bind the application to a certain CPU core or GPU core to avoid the potential fluctuations caused by scheduling. However, we do not have such privileged control on client devices. As a user space application, the controllability over the user's system is limited. Even if an expert user pins the application to a certain core, for commercial systems such as Windows, pinning does not indicate isolating the core for that application only [36]. The system can only ensure the pinned application to run on that core, but could also schedule other processes if still available. Moreover, since our application is not CPU-intensive most of the time, there would usually be idle resources on the same core where the user binds the application to. Therefore, there could still be the same issue of latency increases *at tail*.

C.1.2 Practical Concerns

Since the frame-rate needs to be adjusted at the server, a straightforward concern is whether the frame-rate adaption over the Internet is timely for the stringent delay requirement of high-quality RTC. The measurements in production have two following findings. On one hand, the round-trip network delay is short enough to enable timely feedback: the average round-trip network delay is around 20ms of our cloud gaming service (Appendix C.2.2). Measurements over other high-quality RTC services (e.g., Google Stadia) have similar results of less than 20ms [78, 197]. On the other hand, the degradation of decoding delay usually lasts for a long time, with a median duration of more than 100 milliseconds (Appendix C.2.5). Moreover, we also demonstrate that the increase in decoding delay and net-

CPU	Release date	Score	Portion
Intel® Core™ i5-4590	Q2 2014	868	1.66%
Intel® Core™ i5-7200U	Q4 2016	481	1.61%
Intel® Core™ i5-9400F	Q1 2019	1058	1.56%
Intel® Core™ i5-4460	Q2 2014	801	1.41%
Intel® Core™ i5-5200U	Q4 2014	573	1.38%

Table C.1: Top 5 CPU models of clients in our cloud gaming service.

GPU	Release date	Score	Portion
Intel® UHD Graphics 630	Q3 2017	888	4.54%
Intel® HD Graphics 4600	Q2 2013	474	3.42%
Nvidia GeForce GTX 1050Ti	Q4 2016	5059	3.19%
Intel® HD Graphics 630	Q3 2016	825	2.77%
Nvidia GeForce GT730	Q2 2014	863	2.48%

Table C.2: Top 5 GPU models of clients in our cloud gaming service.

work delay is hardly correlated (Appendix C.2.6). Therefore, for high-quality RTC, when the decoder fluctuates, it is timely enough to control the frame rate over the Internet.

C.2 Measurement over Dataset

In this section, we supplement the observations in the main text with measurements in production. The measurement settings follow the details in §6.5.2.

C.2.1 User Characteristics

In addition to the distribution in §6.3.1, we present the top-5 models, with their release dates, benchmark scores, and portion in our users, of CPU and GPU in Table C.1 and C.2.

C.2.2 Delay Distributions

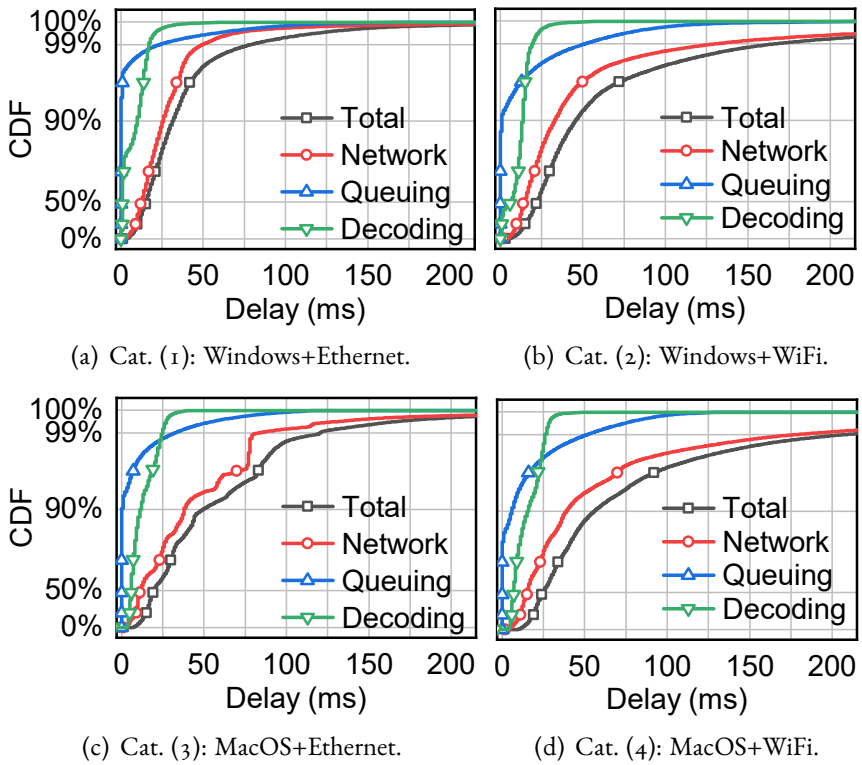
Compared to traditional RTC scenarios, the delay distribution for high-quality RTC has some unique features according to our measurements. We present the Cumulative distribution function (CDF) of component delays and the total delay to explore the delay patterns.

First, due to the edge deployments, the network delay in our cloud gaming service is quite small. According to Figure C.1, the average round-trip network delay is approximately around 20ms. Even in this case, similar to traditional RTC services, the network delay is accounted for a large part of the total delay, the network delay line closely follows the total delay at the median for all four categories in Figure C.1.

However, the tail delay of others component delays like decoding delay and queuing delay are noticeable under cloud gaming scenarios. For the decoding delay, we can notice that the decoding delay for 1080p frames is 18ms at the 99th percentile. Note that the decoder of all sessions evaluated in this paper has been hardware-accelerated. Therefore, as analyzed in §6.3.1, the queuing delay is becoming noticeable at the tail. Referring to Figure C.1, the 99th percentile of queuing delay can reach 50ms under categories (2) and (4), which could degrade users' experience for high-quality RTC services. We further present the root cause analysis below in Appendix C.2.3.

C.2.3 Root Cause Analysis

The total delay is mainly contributed by the network delay, decoding delay, and queuing delay §6.3. Therefore, we want to investigate how these three components contribute to the increase in total delay at the tail. For each frame, we denote T as total delay and C as



(a) Cat. (1): Windows+Ethernet.

(b) Cat. (2): Windows+WiFi.

(c) Cat. (3): MacOS+Ethernet.

(d) Cat. (4): MacOS+WiFi.

Figure C.1: Raw measurements of delays from production.

component delay, where the component delay could be the network, decoding, or queuing delay.

To analyze the necessity and sufficiency of the component delay increasing to the total delay at the tail, we then calculate two conditional probabilities between the event of T longer than a certain threshold T_{th}, and the event of C longer than a certain threshold C_{th}:

- $P(C > C_{th}|T > T_{th})$. We want to account for how component delay increasing contributes to total delay under different delayed degrees T_{th}, and this conditional probability is subject to quantify it. If this conditional probability is close to one,

	Network	Queuing	Decoding
$P(C > C_{th} \mid T > T_{th})$	44.7%	**56.6%**	4.0%
$P(T > T_{th} \mid C > C_{th})$	29.8%	**69.5%**	84.2%

Table C.3: Conditional probabilities with $T_{th} = 100ms$ and $C_{th} = 50ms$ for wired connections, which accounts for 82% of total users of our cloud gaming service.

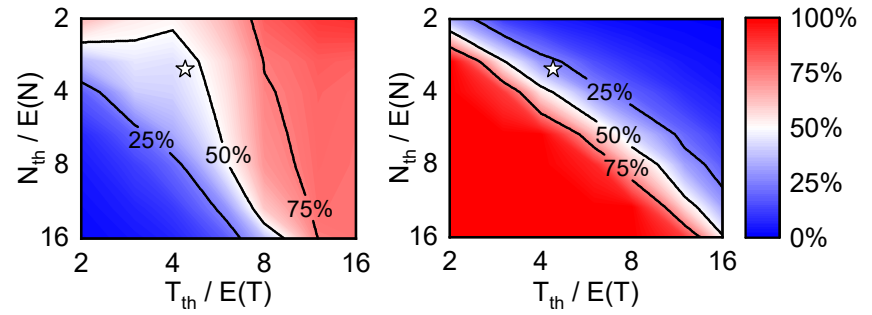
there will be great confidence to blame the component delay for contributing C_{th} delay to the total delay to reaching T_{th}.

- $P(T > T_{th} \mid C > C_{th})$. As the sum of component delays, the total delay should increase when one of the component delays increases. This conditional probability is subject to illustrate this assumption and indicates the probability of total delay reaching the T_{th} under different component delay increasing degree C_{th}.

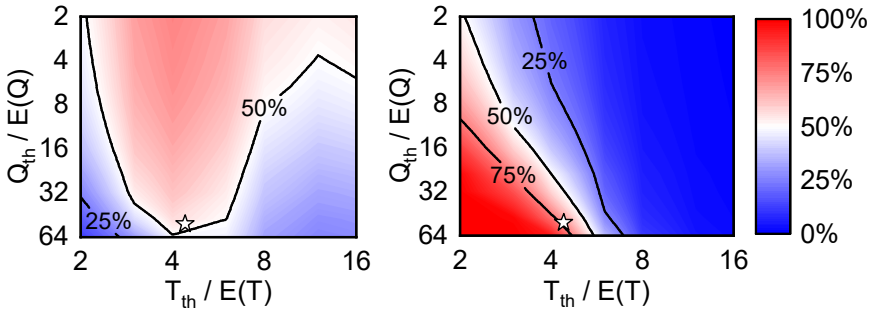
We calculate the conditional probabilities for three components for different C_{th} and T_{th}, and have the following observations.

Total delay increasing is a reflection of components delay increases. As the sum of the different types of components delay, It's obvious that no matter what kind of component delay is increasing, the total delay will also increase.

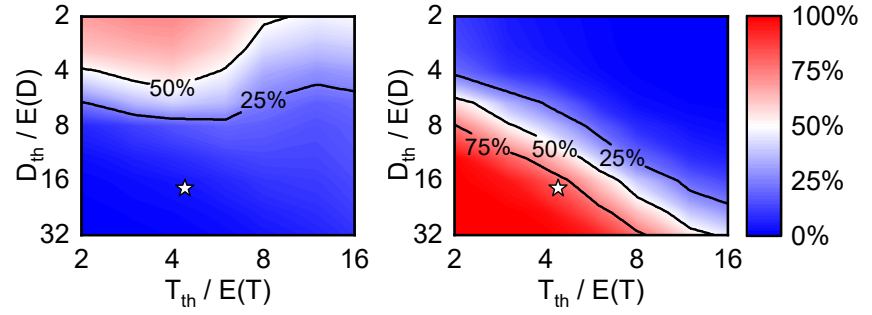
So to find out the sufficiency of total delay increasing, we calculate the conditional probability of $P(T > T_{th} \mid C > C_{th})$ in right-side of Figure C.2. We can notice that for all the component delays, their delay increasing can also mean a higher probability of total delay increasing (75%ile line in the figure is shifting to the right with the component delay increasing). The down-left corner is 100%, because as the sum of all types of component delay, the total delay must be larger than any component delay.



(a) $P(N > N_{th}|T > T_{th})$. (b) $P(T > T_{th}|N > N_{th})$.



(c) $P(Q > Q_{th}|T > T_{th})$. (d) $P(T > T_{th}|Q > Q_{th})$.



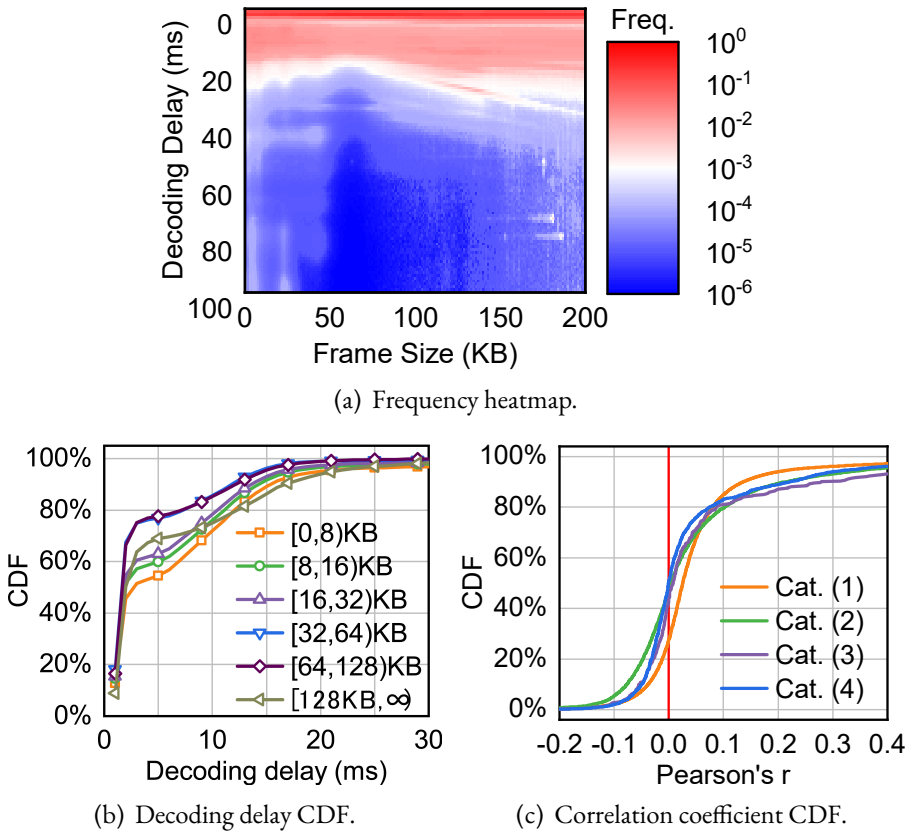
(e) $P(D > D_{th}|T > T_{th})$. (f) $P(T > T_{th}|D > D_{th})$.

Figure C.2: The heatmap of conditional probabilities for wired connections. The horizontal and vertical axes have been normalized by their average values. The star point's value is recorded in table C.3 The down-left corner is 100% since the total delay should always be larger than the component delay.

Queuing delay is responsible for delay increases of >100ms. To figure out the necessity of total delay increasing, we calculate the conditional probability of $P(C > C_{th}|T > T_{th})$ in left-side of Figure C.2. Our major finding is that with the different order of severity of total delay increasing (2-16× of $\mathbb{E}(T)$, the root cause of it is also changing. As we can see, when T_{th} is larger than $8\mathbb{E}(T)$, network delay has a high probability (shaded red) to be blamed. However, when T_{th} is from $3\mathbb{E}(T)$ to $8\mathbb{E}(T)$, queuing delay dominates the most increased events. It illustrates that the queuing delay is responsible for the increase of total delay by around 100ms. Specifically, we present the conditional probabilities for three components with $T_{th} = 100ms$ and $C_{th} = 50ms$ for wired connections in Table C.3. As we can see, queuing delay has both high $P(C|T)$ and $P(T|C)$. Indicating that the total delay has a great possibility of reaching 100ms when queuing delay increases to 50ms. And for those video frames that total delay truly getting the 100ms, there will be great confidence to blame the queuing delay for contributing to the majority of delay increasing. So the queuing delay will be the root cause of the increase of total delay to 100ms.

C.2.4 Decoding Performance

In this section, we explain the reasons behind the ineffectiveness of controlling the service process for eliminating queuing time by adjusting the bit rate. The decoding time of decoders mainly depends on the resolution of the streaming. However, due to the dependency between frames, changing the resolution during the streaming will make the subsequent frames undecodeable and needs to request a new key frame for most codecs [87]. Yet, since the frame size of key frames is usually several times ofthose of other frames [158], fre-



(a) Frequency heatmap.

(b) Decoding delay CDF.

(c) Correlation coefficient CDF.

Figure C.3: The correlation between the frame size and decoding delay for hardware decoders.

quently requesting key frames will impose additional overhead on the network and degrade the users' experiences.

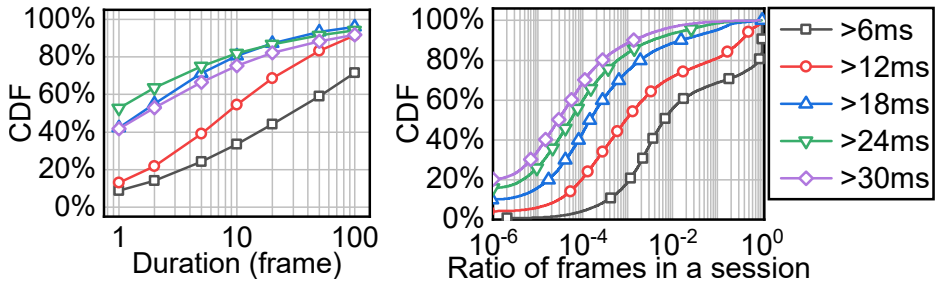
Another straightforward solution is to try to accelerate the service process by reducing the bit rate while maintaining the same resolution. With the same resolution and frame-rate options, reducing the bit rate means lesser video data per video frame can carry. We are to investigate whether sending video frames with smaller data sizes is helpful for decoding acceleration. However, according to our measurements on the H.264 decoder, merely changing the bit rate does not significantly reduce the decoding time.

We measure the relationship between the frame size and decoding time of the dataset described in §6.5.2. We first present the heat map in Figure C.3(a). With the variation of frame size, the distribution of decoding time does not significantly change, where the decoding time of most frame sizes intensively falls around several milliseconds, as shown in the red area at the top of the heat map. To eliminate the frame size variation under the same target bit rate, we split the frame size into different intervals and present the cumulative distribution function (CDF) in Figure C.3(b). As the frame size become larger, the [128KB, ∞) the line does not locate in the rightest area (higher decoding delay). And other frame size interval's CDF lines stay together, indicating that the lowering frame size does not help for the decoding time acceleration.

Moreover, we split the dataset into four different categories (Table 6.1), to demonstrate that reducing frame size will not help decode acceleration under various platforms. We leverage the Pearson correlation coefficient to illustrate the independence, which value of zero can indicate that there is no association between the two variables [236]. Figure C.3(c) shows that most of the Pearson's r value is located around zero, indicating the poor association between frame size and decoding delay. Therefore, controlling the service process of encoding bit-rate cannot effectively reduce the decoding time and alleviate the load of the decoder queue.

C.2.5 Decoder Degradation

Because the queue overhead will be introduced by the mismatch of the rate of two sides of the queue [133], if the decoding speed is not capable of processing the incoming default 60fps, it will be necessary for AFR to change to a lower target frame rate. However, since



(a) Decoder degradation duration.

(b) Frames with long decoding delay

Figure C.4: Decoder degradation when filtered with different thresholds for decoding delay.

the client and server are located distant, the frame-rate adjustment request to the server side will need a control loop to take effect on the client side with the updated frame rate. So if the AFR control loop is shorter than the decoder degradation duration, the decoder will be capable of processing a higher incoming frame rate before the AFR requests take effect.

We measure the duration of the decoder degradation level over the traces introduced in §6.5.2. As we can see in Figure C.4(a), for frames with a decoding time of more than 12ms, 50% of them last for more than 10 frames. Under 60fps streaming, considering the average of RTT is close to one frame interval of 16.7ms, and the 90%ile encoder response delay is less than three frames interval §6.6.4. In this case, lowering the frame rate will be helpful for alleviating the decoder queue even under the control loop delay of AFR. Therefore, AFR is capable of timely adjusting the frame rate to adapt to the decoder degradation. Moreover, the AFR can significantly help alleviate the queue overhead under those frames with a long period of decoder degradation and sustain queuing time for waiting for overhead queue elimination.

We further measure the ratio of frames with different decoding delays and present the results in Figure C.4(b). Half of the user sessions suffer from a decoding delay of >12ms for

at least 1‰ frames. This also indicates that the degradation of decoding delay is a general issue among all clients.

C.2.6 Component Correlation Analysis

The streaming pipeline will be affected by many components, like the networking, decoding, and queuing delays can both cause total delay increases to degenerate the user's experience Appendix C.2.3. In this paper, we propose AFR to reduce the tail queuing delay by matching the arrival rate of the decoder queue to the service rate (decoding speed). When decoding delay increases to disable decode frames timely, the AFR will send a frame-rate adjustment request from the client to the server. However, the request and subsequent frames need to be transported through the network. Therefore, a straightforward question is: does the increase of decoding delay affect the network delay to put an extra effect on the AFR control loop? We will figure out this by measuring the independence of those component delays.

We quantify the independence of different component delays with Pearson's r value [236], dynamic time warping (DTW) [61], and Cramer's v value [90]. In short, all these metrics demonstrate the poor association between networking and decoding delay, inclining that we could decouple the network and decoder issues and independently control them.

Regarding the Pearson correlation coefficient, the value of zero can indicate that there is no correlation between the two variables [236]. Figure C.5 illustrates that for all four categories in Table 6.1, the Pearson's r value of networking and decoding are close to zero, indicating a poor correlation between them.

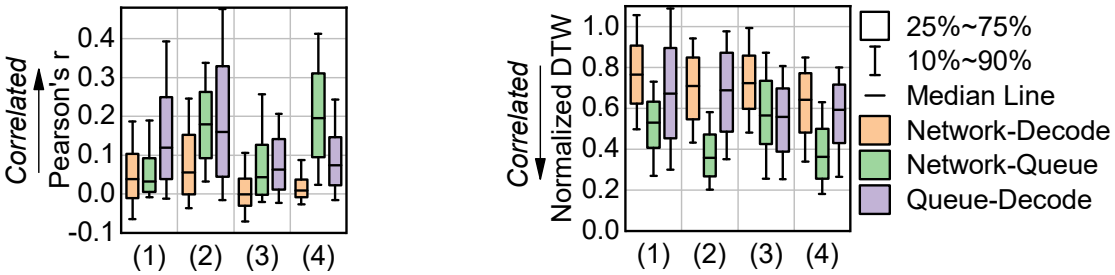
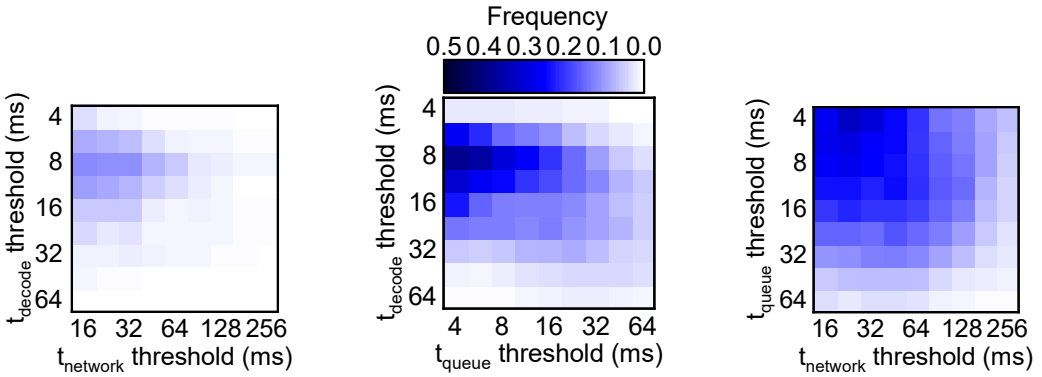


Figure C.5: Pearson's r (left, higher is more correlated) and normalized DTW distance (right, lower is more correlated) between delay components.



(a) Network-Decode (Max: 0.12). (b) Queue-Decode (Max: 0.39). (c) Network-Queue (Max: 0.32).

Figure C.6: Cramer's V between different delay components.

Moreover, the different component delays might be correlated with each other across frames. For example, the decoding delay could affect the subsequent queuing delay by its incapacity to decode video frames timely. To measure the correlation across frames, we leverage DTW to calculate the optimal match between two-time series [61]. The DTW algorithm is widely used in many scenarios like sign language recognition and time series clustering [159, 204]. The optimal match calculation under DTW is denoted by the match with minimal cost, where the cost is computed as the sum of absolute differences, for each matched pair of indices, between their values. Therefore, a larger DTW distance can be considered the mismatch between two series to a further extent. According to Figure C.5,

the normalized DTW distance of network delay to decoding delay under all four categories is large, showing the lack of correlation between them.

The strength of the relationship can also be assessed by Cramer's V value, which is a metric based on the χ^2-test but normalized for different data sizes. It indicates how strongly two categorical variables are associated [90]. A Cramer's V value of $\leqslant$ 0.1 can be interpreted as hardly correlated [72]. According to our measurement in Figure C.6, we can notice that all the Cramer's V values of networking and decoding delay are $\leqslant$ 0.2, illustrating the weak association between networking and decoding state. Therefore, according to our measurements before, we can see the independence between networking and decoding delay.

C.3 Simulator Implementation

In this section, we introduce the implementation of our simulator. Specifically, traces are recorded in the following format:

$$R(n) = \left[ts^{(n)}, \tau_{net}^{(n)}, \tau_{queue}^{(n)}, \tau_{decode}^{(n)}\right] \tag{C.1}$$

where $ts^{(n)}$ is the arrival timestamp of the n-th frame, τ_{net}, τ_{queue}, and τ_{decode} are the round-trip network delay, queuing delay, and decoding delay of that frame. The simulator reads the traces frame-by-frame at specific timestamps and measures the current frame rate based on the interarrival time as §6.4.2. The simulator then dequeues the head frame in the decoder queue when the decoder is available, where the decoding time of each frame is also read from the trace.

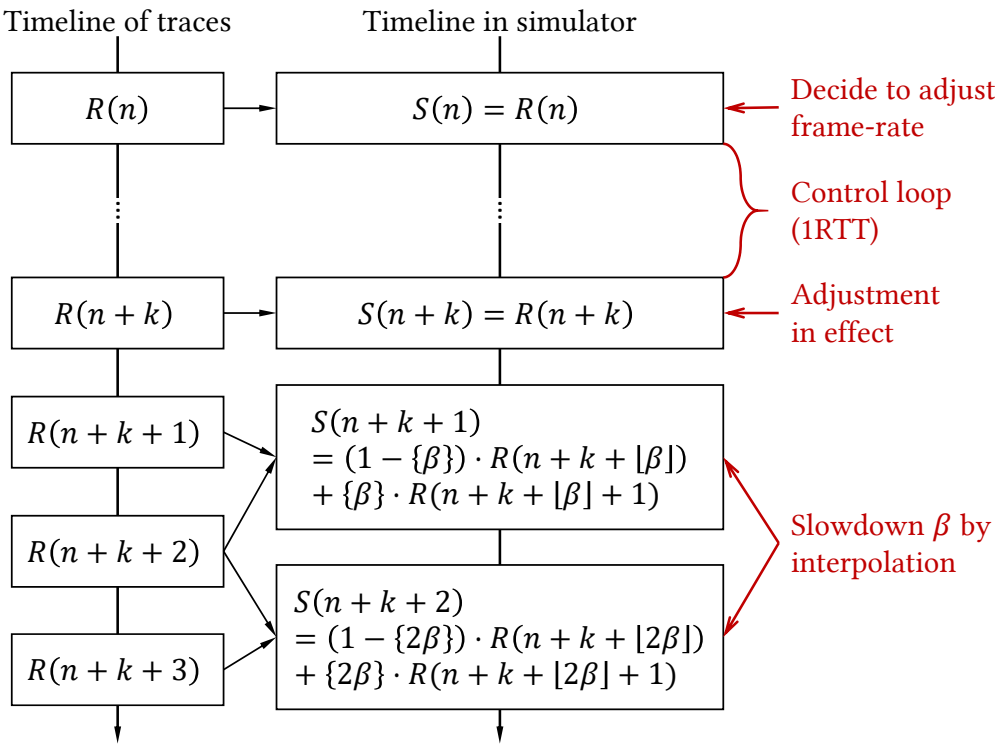


Figure C.7: Illustration of frame-rate adjustment in our simulator.

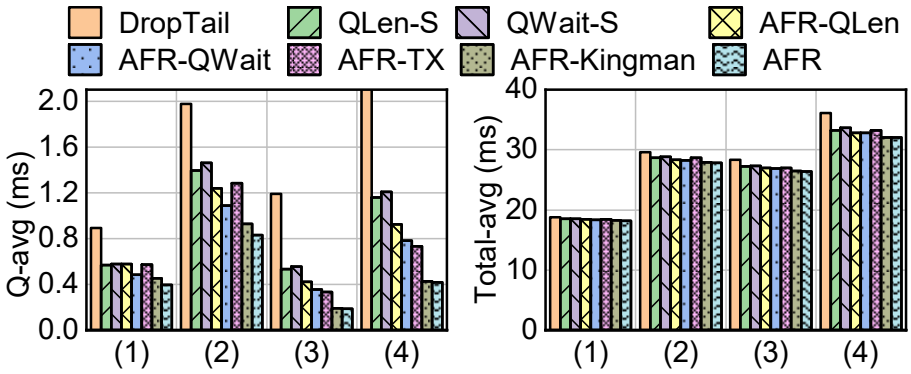
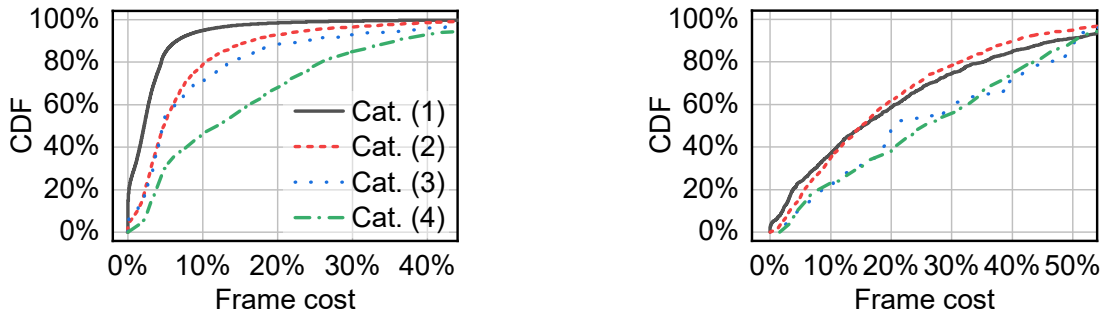


Figure C.8: Average queuing delay (left) and total delay (right).

When the adaptive frame-rate decides to set the frame-rate to f_{set}, the simulator first reads the current control loop by the round-trip network delay of the current frame $\tau_{net}^{(n)}$. The simulator then calculates the earliest frame $n + k$ that the new frame-rate f_{set} will take in effect:

$$k = \arg\min_{k} \left(ts^{(n+k)} - ts^{(n)} \geqslant \tau_{net}^{(n)} \right) \tag{C.2}$$

After that, based on the measurement of the current frame-rate f_{cur}, the simulator calculates the slowdown factor $\beta = f_{cur}/f_{set}$, and reads the traces with a slowdowned speed. For example, as shown in Figure C.7, When there are frames $R(n+k+1)$ to $R(n+k+3)$ in the original trace, the simulator reads the traces with indices $R(n+k+\beta), R(n+k+2\beta), \cdots$. When β is not integer, the simulator interpolates the traces with its neighbor frames ($S(n+k+1)$ and $S(n+k+2)$).



(a) All sessions.

(b) Stuttered sessions.

Figure C.9: The number of wasted frames when skipping frames instead of adjusting the frame rate for AFR.

C.4 Supplementary Experiments

C.4.1 Average Delay

We further measure the average queuing delay and total delay for four traces and present the results in Figure C.8. As we can see, the reduction of tail delay of AFR does not sacrifice the average delay on all traces. In contrast, the average delay has also been slightly improved against baselines, due to the improvements at the tail.

C.4.2 Frame Costs of AFR with Skipping

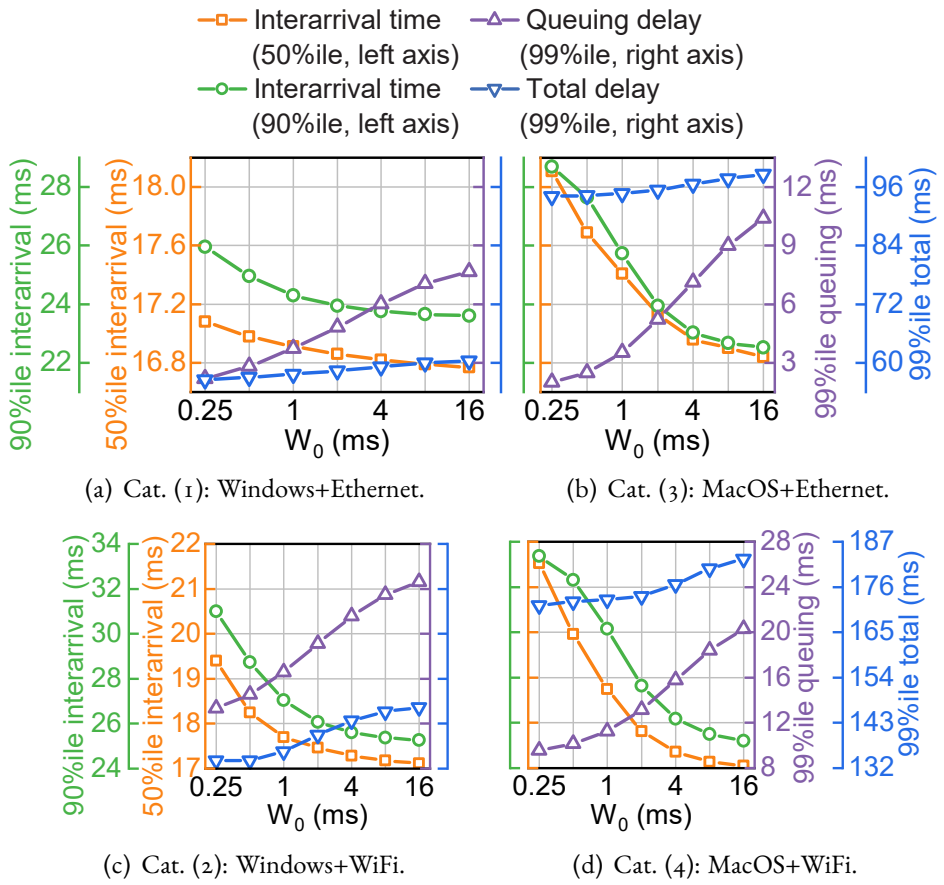
Besides, as we discussed in §6.6.4, skipping frames without changing the frame rate from the content generator (e.g., gaming application) would waste the rendering resources of the server. For example, for high-quality RTC, rendering at 60fps would take approximately one time more GPU resources than rendering at 30fps. Therefore, we measure how many frames have been wasted (i.e., frame cost) if we merely skip the frames to approximate the target frame rate without adapting the content generator.

We present the results of all traces in Figure C.9. For all traces, adjusting the frame rate could save 3% to 12% frame costs in all traces, saving considerable operating expenses for the service provider since GPU is one of the highest expenses. For stuttered sessions (following the definition in §6.6.2), the saved frame cost would be even higher.

C.4.3 Parameter Sensitivity

Long-term control target (W_0) We present the simulation results on the sensitivity of W_0 (in the stationary controller) on different traces in Figure C.10. As we discussed in §6.5.2, a lower W_0 results in a more aggressive queue control yet leads to the degradation of frame rate. We vary W_0 from 0.25ms to 16ms and measure the interarrival time, queuing delay, and total delay. By adjusting W_0, operators could effectively balance the total delay and frame rate. Therefore, operators could adjust W_0 according to the preferences on total delay and frame rate for different users and games.

EWMA discounting factors (ξ_{arrv} and ξ_{serv}). We also vary the EWMA discounting factors (ξ_{arrv} for the arrival process and ξ_{serv} for the service process). Higher ξ indicates that the EWMA focuses on the recent values more to capture changes, while a lower value indicates more attention to the historical trends. As shown in Figure C.11, the performance metrics (including the queuing delay, total delay, and frame rate) are relatively robust to these two parameters. By varying ξ_{arrv} and ξ_{serv} across several magnitudes, most metrics change marginally. For example, the 99%ile of queuing delay changes by $4\times$ when varying W_0 (Figure C.10) while only changes by less than 15% when varying ξ_{arrv} by three magnitudes (Figure C.11). We also observe trends in varying ξ_{arrv} and ξ_{serv}. Lower ξ_{arrv} values



(a) Cat. (1): Windows+Ethernet.

(b) Cat. (3): MacOS+Ethernet.

(c) Cat. (2): Windows+WiFi.

(d) Cat. (4): MacOS+WiFi.

Figure C.10: Sensitivity analysis on W_0 on different traces.

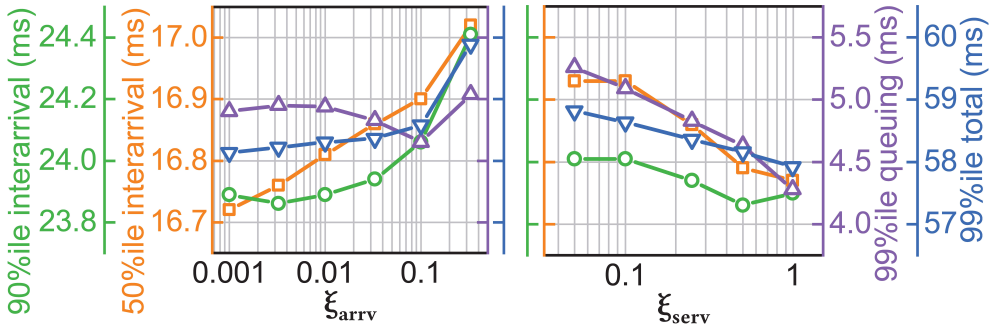


Figure C.11: Performance of AFR with different settings of ξ_{arrv} and ξ_{serv}. Y-axes have been magnified compared to Figure C.10.

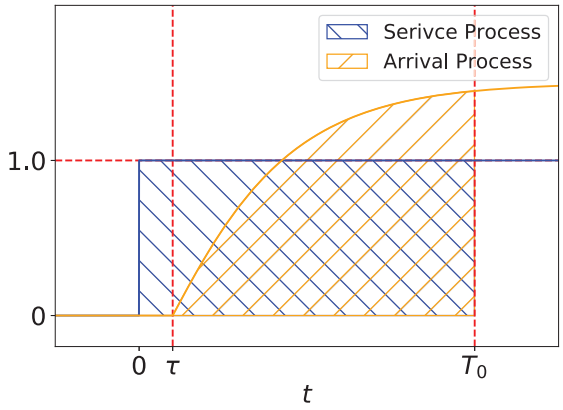


Figure C.12: The system begins to control the queue after control-loop delay τ and stabilize the queue at T_0.

will slightly improve the performance of AFR, implying that the long-term behavior of arrival service is more critical. Higher ξ_{serv} also slightly improves the performance, indicating focusing on recent decoding time is helpful. This is because we have already filtered out outlier decoding time. Paying more attention to recent decoding time could make the AFR quickly adjust the frame rate.

C.5 Convergence Analysis

Finally, we provide a detailed analysis of the convergence time during the state transitions of the stationary controller. As introduced in §6.4.2, let the expectation of queuing delay $E(\tau_{queue}) = W_0$, according to Eq. 6.1, we have:

$$\mu_a = \frac{\mu_s}{\rho} = \left(1 + \frac{c_a^2 + c_s^2}{2W_0}\mu_s\right)\mu_s \tag{C.3}$$

Then we can discuss the *convergence time* of the system. The convergence time here refers to the time at which the stationary controller converges to a stationary state when

the service process changes, and the potential accumulated queue during the transition is drained up.

Specifically, without loss of generality, we discuss a simplified case shown in Figure C.12: Both the arrival and service process have an average value of zero for $t < 0$, and the service process changes from zero to one at $t = 0$. The arrival rate will gradually respond to the change after a control loop of τ. We want to find the convergence time T_0 where

$$\int_0^{T_0} \mu_a \, \mathrm{d}t > \int_0^{T_0} \mu_s \, \mathrm{d}t \tag{C.4}$$

In this case, the queue accumulated during the response to the arrival rate will be cleared. We further illustrate the convergence in Figure C.12. By substituting Eq. C.3, we have:

$$\int_\tau^{T_0} \left(\mu_s + \frac{c_a^2 + c_s^2}{2W_0} \mu_s^2 \right) \mathrm{d}t > \int_0^{T_0} 1 \, \mathrm{d}t \tag{C.5}$$

From the measurement of EWMA in Eq. 6.5, we have

$$\hat{\mu}_s = 1 - (1 - \xi_\mu)^{t-\tau} \quad (t > \tau) \tag{C.6}$$

Therefore, let $\gamma = 1 - \xi_\mu$ to simplify the expression, we need to find the minimum T_0 such that:

$$\int_0^{T_0-\tau} \left((1 - \gamma^t) + \frac{c_a^2 + c_s^2}{2W_0} (1 - \gamma^t)^2 \right) \mathrm{d}t > T_0 \tag{C.7}$$

By solving the integral in Eq. C.7, finally we have

$$W_0 < \frac{c_a^2 + c_s^2}{2} \frac{(\gamma^{T_0-\tau} - 1)(\gamma^{T_0-\tau} - 3) + 2(T_0 - \tau)\ln\gamma}{2(\gamma^{T_0-\tau} - 1) + 2\tau\ln\gamma} \tag{C.8}$$

For example, when set $c_a^2 + c_s^2 = 2$, we vary the other parameters in Eq. C.8 and present the minimum T_0 in Figure C.13. In the most general settings of AFR ($\tau = 1$ since the average RTT is around 15ms, $\xi_\mu = 0.25$ as introduced in §6.5.2, $W_0 = 2$ms), the stationary

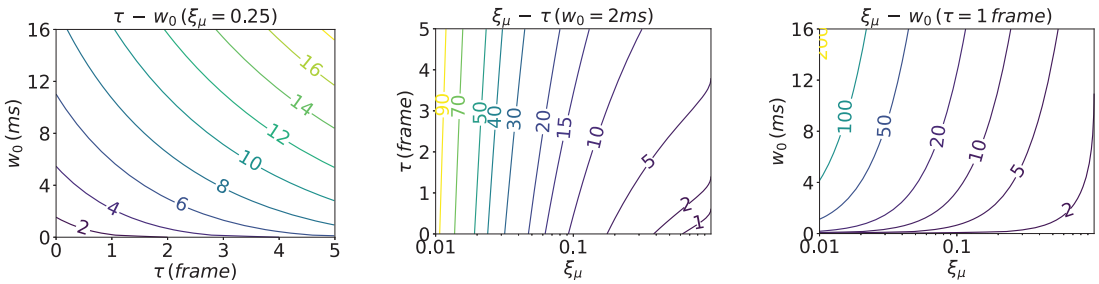


Figure C.13: Contour plot of the convergence region of T_0 with different parameters.

controller can converge to the new stationary state within 2 frames. In other settings of the AFR parameters, the stationary controller could also converge and drain the queue within tens of frames, which is much less than the frame-rate adjustment interval of hundreds of frames as evaluated in §6.6.2.

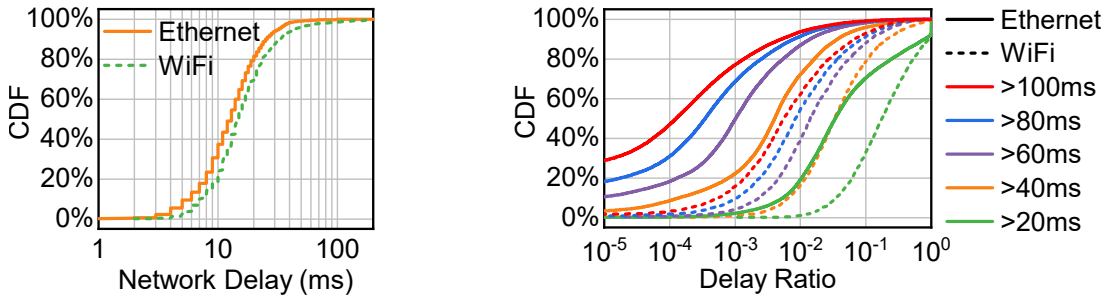
D

Hairpin (§7)

D.1 Measurements in Production

We present our measurement results on the cloud gaming service X in production to support some claims in the paper.

To investigate the effect of edge acceleration of interactive streaming in the wild, we conduct a measurement campaign on the cloud gaming service X. The measurements last



(a) Frame-level delay

(b) Session-level delay ratio

Figure D.1: Network delay distributions of the interactive streaming service of company T. Delay ratio is the ratio of frames with a delay of $> 20, > 40, > 60, > 80$ and > 100 ms in each session. Note that the delay here is measured at the application layer (details in §7.4.2).

for one week with thousands of sessions (containing heterogeneous users through Ethernet, WiFi with Windows and MacOS systems) and are presented in Fig. D.1. As shown in Fig. D.1(a), the majority of network delay collected at the granularity of video frame falls into 10-20ms for both Ethernet and WiFi. We also measure the flow-level delay ratio at different thresholds and present the results in Fig. D.1(b). With the edge acceleration, the ratio of frames with longer than 100ms delay in most flows is less than 10^{-2}. Among them, Ethernet flows perform slightly better than WiFi flows. This validates the effectiveness of edge acceleration: the average network delay could be reduced to 10-20ms with a proper edge acceleration.

We further measure the fluctuation of RTT by the duration when RTT is roughly kept at the same level. We quantify it by calculating the transmission chance (i.e., layer L) for the RTT measured by each frame, and calculate the duration when the chance is kept the same. For example, given a deadline of 100ms in this paper, when the RTT measurements are [26ms, 18ms, 17ms, 22ms, 17ms, 19ms, 19ms], the transmission chances are [3, 5, 5, 4, 5, 5,

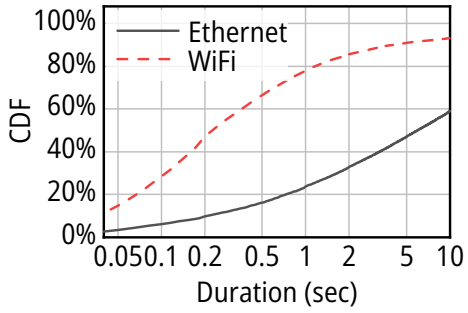


Figure D.2: Distribution of network RTT maintenance duration in our interactive streaming service.

5]. In this case, the durations of each transmission chance are [1, 2, 1, 3], which are denoted as *RTT maintenance durations*. We present the distribution of RTT maintenance duration measured in our cloud gaming service in Fig. D.2. The RTT maintenance duration of Ethernet is much longer than that of WiFi, indicating that Ethernet has a more stable end-to-end delay. Meanwhile, the median duration of both Ethernet and WiFi is above hundreds of milliseconds, which is much higher than the feedback loop of Hairpin. This indicates that *RTT does not frequently change, and* Hairpin *is able to detect and react to the fluctuations of RTT.*

D.2 Optimization Model

In this section, we present the notations used in the Markov chain in §7.3.3. We further present the detailed designs here.

D.2.1 Optimization of the Redundancy Rate

We build an absorbing Markov chain to model the redundancy and calculate the deadline miss rate considering retransmission, as shown in Fig. 7.6. We first define the *state* in

the Markov chain as (n, r), where n is the number of unacknowledged packets within the block, and r is the number of retransmission. For example, $(d_0, 0)$ represents the initial transmission where all d_0 packets have not been received before (since it is the first time of transmission). $(3, 2)$ denotes that there are still 3 data packets that need to be retransmitted for the second time.

We first calculate the transition probability between states in the Markov chain. For the transition between state (n_1, r) to $(n_2, r-1)$, we know that n_2 data packets are lost in the r-th transmission and need to be transmitted for the $(r+1)$ time. We first discuss the scenario of $n_2 > 0$. We denote the total number of packet losses (including data and redundancy) in the r-th transmission as $l(n_1, r)$. We denote the number of redundant packets in the r-th transmission as $k(n_1, r)$. Since the packet losses of all packets should not be less than the packet losses of data packets, we have $l(n_1, r) \geqslant n_2$. Meanwhile, since there are only $k(n_1, r)$ redundant packets in total, we have $l(n_1, r) \leqslant n_2 + k(n_1, r)$. We also have $l(n_1, r) > k(n_1, r)$, otherwise the lost packets could be recovered with FEC. Therefore, the probability of n_2 data packet losses under the condition of $l(n_1, r-1)$ total packet losses follows the hypergeometric distribution:

$$\begin{aligned} &H(n_2; n_1 + k(n_1, r-1), n_1, l(n_1, r)) \\ &\quad = \binom{n_1}{n_2}\binom{k(n_1, r)}{l(n_1, r) - n_2} \Big/ \binom{n_1 + k(n_1, r)}{l(n_1, r)} \end{aligned} \tag{D.1}$$

Thus, the transition probability from (n_1, r) to $(n_2, r-1)$ is:

$$p((n_1, r) \to (n_2, r-1)) = \sum_{l(n_1,r)} H(n_2; n_1 + k(n_1, r), n_1, l(n_1, r)) \cdot P(l(n_1, r) \text{ losses}) \tag{D.2}$$

On the other hand, at the loss rate of α, losing $l(n_1, r)$ packets in all $n_1 + k(n_1, r)$ packets follows the Binomial distribution:

$$P(l(n_1, r) \text{ losses}) = Bi(l(n_1, r); n_1 + k(n_1, r), \alpha) = \binom{n_1 + k(n_1, r)}{l(n_1, r)} \alpha^{l(n_1,r)} (1-\alpha)^{n_1 + k(n_1,r) - l(n_1,r)} \tag{D.3}$$

Therefore, by substituting Eq. D.1 and D.3 into Eq. D.2, we can have the transition probability for $n_2 > 0$. Similarly, when state transits from (n_1, r) to $(0, r-1)$, then the number of lost packets in the r-th layer of Fig. 7.6 must be less than $k(n_1, r)$. Therefore, the transition probability satisfies:

$$p((n_1, r) \to (0, r-1)) = \sum_{i=0}^{k(n_1,r)} Bi\left(i; n_1 + k(n_1, r), \alpha\right) \tag{D.4}$$

D.2.2 Optimization of Block Size

In the following analysis, we are going to compare the utility of transmitting the whole frame for L chances, or splitting the frame into several blocks and some of them enjoying $L+1$ chances. With that, we assume that the dispersion is less than one RTT.

Therefore, when the block size is set to d, there are N_{L+1} blocks that could enjoy $L+1$ chances of transmission, and the remaining N_L blocks with L chances of transmission,

where

$$\begin{aligned} N_{L+1} &= \left\lfloor \frac{DDL - (L+1) \cdot RTT}{d/\Theta} \right\rfloor \\ N_L &= \left\lceil \frac{F}{d} \right\rceil - N_{L+1} \end{aligned} \tag{D.5}$$

Therefore, the on-time delivery of the frame requires the on-time delivery of each block. Since the deadline miss rate is equal to one minus the probability of on-time delivery, we have the frame DMR (FDMR) given a certain block size d as:

$$\begin{aligned} 1 - FDMR(d) &= (1 - DMR(L+1, d))^{N_{L+1}} \cdot (1 - DMR(L, d))^{N_L} \\ \Rightarrow FDMR(d) &= 1 - (1 - DMR(L+1, d))^{N_{L+1}} \cdot (1 - DMR(L, d))^{N_L} \\ &= N_{L+1} \cdot DMR(L+1, d) + N_L \cdot DMR(L, d) \end{aligned} \tag{D.6}$$

where the last equation holds since $DMR(L, d) \ll 1$ and $(1 - \alpha)^n = 1 - n\alpha$ when $\alpha \ll 1$. As for the bandwidth cost, recalling Eq. 7.5, the number of extra packets of the frame is the sum of the number of extra packets for each block. Since the BWC of each block shares the same denominator (frame size S), the frame BWC is also the sum of BWC of each block:

$$FBWC(d) = N_{L+1} \cdot BWC(L+1, d) + N_L \cdot BWC(L, d) \tag{D.7}$$

Therefore, the optimal block size is:

$$d_{opt} = \arg\max_{d} \ utility\left(FDMR(d), FBWC(d)\right) \tag{D.8}$$

In our implementation, we iterate the possible block size B from 1 to the frame size S, and store the optimal block size in each scenario in an offline lookup table. Since the $DMR(L, B)$ and $BWC(L, B)$ are accessible in the absorbing Markov chain constructed above, the construction of the table is time-efficient.

D.3 Implementation Details

We are going to introduce the sending mechanism beneath Hairpin and the implementation of the redundancy optimization in Hairpin.

Acknowledgement aggregation. In wireless networks, researchers also propose to aggregate several acknowledgements at the client side to alleviate the uplink interference [163]. However, the delayed acknowledgement might also interfere with the measurements of RTT, delay the detection of packet losses and waste potential chances of retransmission. In our implementation, to eliminate the interference from acknowledgement mechanisms, we disable the aggregation of acknowledgements. The precise measurement of RTT in the scenario of aggregated acknowledgement could also be implemented with recent efforts such as TACK [163], which is out of our scope.

Note that this is different from the aggregation on wireless routers [62]. Such aggregations due to wireless channel competition should be reflected in our measurements of network RTT fluctuations in Fig. 7.2. In our simulation with online measurements and deployments in production, Hairpin behaves well even with the RTT fluctuations.

FEC codec. For the scenarios with a redundancy rate of $\leqslant 100\%$, we implement the FEC codec as RS-FEC, as suggested by many other related efforts [220]. We refer the readers

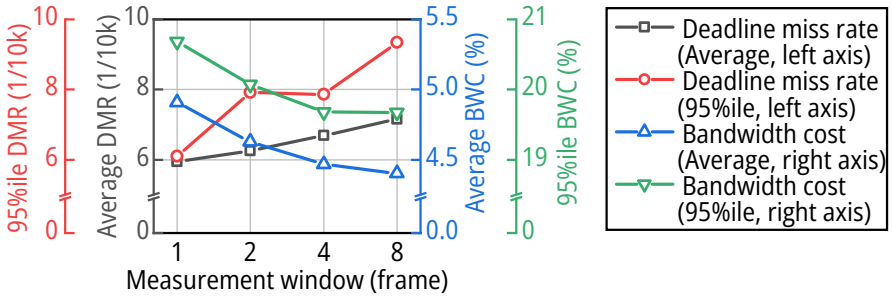


Figure D.3: Sensitivity of the measurement window in §7.3.4.

to [220] for the details of RS-FEC. However, when implementing the redundancy rate of >100%, RS-FEC is not designed to reliably recover lost packets in all cases. For example, when there are 2 data packets and 4 FEC packets, RS-FEC cannot always recover 2 data packets when there are 4 packet losses due to the invertibility of the decoding matrix: it depends on whether two packets received at the client are linearly independent at the generation matrix.

Therefore, we implement a customized FEC codec. For example, for data packets a and b, when considering them as two numbers (with a length of up to 12kbits), we could calculate $a + b$, $a + 2b$, $2a + b$, etc., and send them to the client. The only overhead is the additional bits that could overflow from the addition, which is much less than the data bits. Moreover, as shown in Fig. D.8, in most cases the redundancy rate is less than 100%. Therefore, the overall decoding overhead is also acceptable. We leave the further adoption of advanced FEC codec when the redundancy rate is >100% as our future work.

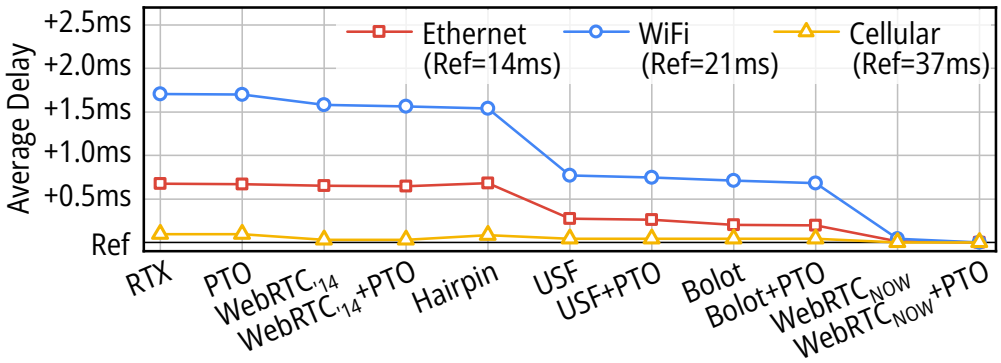


Figure D.4: Average end-to-end delay of in the experiments in §7.4.3. We trim the lowest average delay in different traces for comparison.

D.4 Supplementary Experiments

Measurement window. We also evaluate the performance of Hairpin by adjusting the measurement window of the network conditions that we discussed in §7.3.4. Since Hairpin optimizes the redundancy parameters based on real-time measurements of the network conditions, the size of the measurement window might affect the performance of Hairpin. We vary the measurement window from the last 1 to 8 frames and reconduct the experiments over WiFi traces. We measure the average and 95th percentile DMR and BWC, and present the results in Fig. D.3. The DMR and BWC are quite robust: By varying the measurement window from 1 to 8, the average DMR and average BWC vary within 0.47%-0.49% and 6.94%-7.19%, which is subordinate to the improvements in §7.4.3 (Fig. 7.8(b)). In practice, operators can decide the measurement window based on the fluctuations of network conditions.

Per-frame latency of Hairpin. Besides, we also measure the average end-to-end delay for the successfully delivered frames in the experiments in §7.4.3 for Hairpin and different baselines.

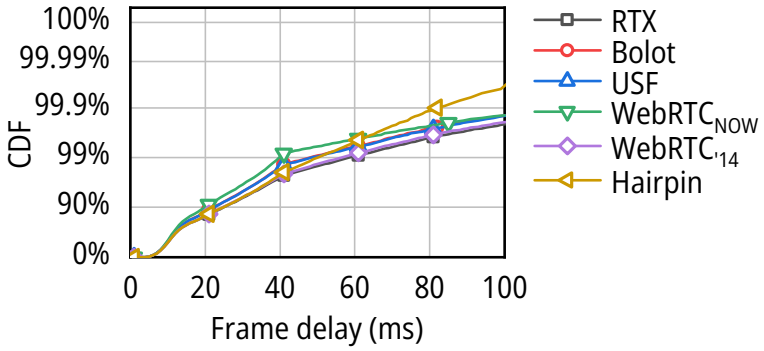
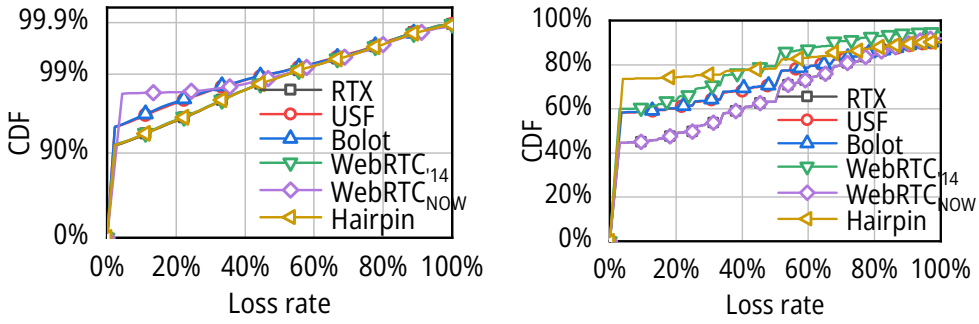


Figure D.5: The distribution of the delivery time of each frame. Note that the y-axis is log-scaled.

As shown in Fig. D.4, the average end-to-end delay of Hairpin does increase compared to the baseline with the lowest average delay. However, the increase is only 0.1-1.5ms for all traces, which is negligible compared with the RTT (1%-7%), and considering the deadline effect we discussed in §7.2.1. Furthermore, operators could also adopt less aggressive mappings (e.g., increasing λ) to tradeoff between the tail delay and average delay.

We also present the distribution of the delay of each frame in Fig. D.5. Similar to Fig. D.4, the average (median) latency of frames of Hairpin is similar to other baselines. However, Hairpin could reduce the tail latency significantly. For example, Hairpin can reduce the 99.9th percentile frame latency to 80ms while all baselines of longer than 100ms. Looking at the vertical axis, Hairpin is also capable of reducing the ratio of higher than 100ms by more than a half, as shown in Fig. 7.8(b).

Loss rates in each round. We further present the distributions of loss rates of all frames in each round (specifically, initial transmission and the third retransmission) in Fig. D.6. This expands the results in Fig. 7.13(b). We can tell from Fig. D.6(a) that due to the conservative redundancy strategy of Hairpin, the loss rate of Hairpin is higher. However, when



(a) Initial transmission

(b) The 3rd retransmission

Figure D.6: Distribution of loss rates by frame in each round of transmission.

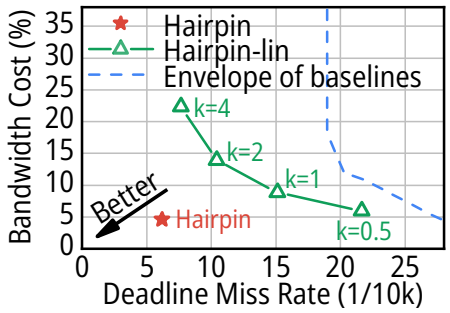
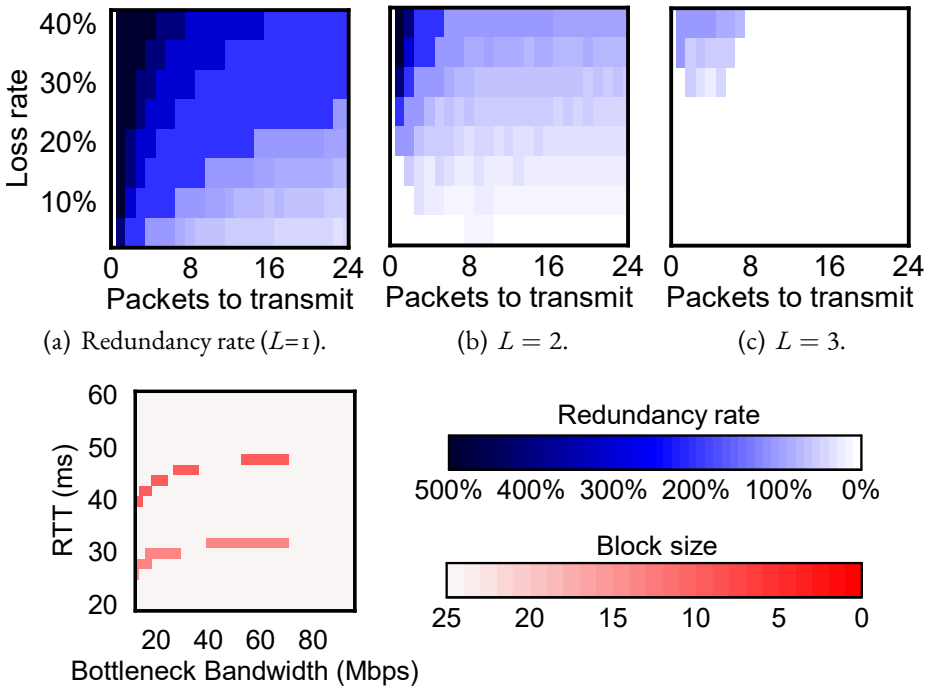


Figure D.7: Heuristic-based Hairpin (Hairpin-lin). The envelope of baselines is from Figure 7.8(b).

retransmission starts, Hairpin is able to maintain a low loss rate – which means a high success rate in delivering frames – compared to other baselines. This shows the strategy of Hairpin: conservatively adding FEC packets when deadline is far away, and aggressively adding FEC packets to retransmissions.

The improvements of using Markov chain. As we analyzed in §7.3.2, a strawman solution is good but not enough to fully utilize the design space of redundancy and retransmission. Thus, we also evaluate the heuristic baseline we present in §7.3.1 (denoted as Hairpin-lin, with sweeping the coefficient k from 0.5 to 4, and present the results in Figure D.7. As



(a) Redundancy rate (L=1). (b) $L = 2$. (c) $L = 3$.

(d) Block size and legends.

Figure D.8: Optimization results by Hairpin. Fig. D.8(a) to D.8(c) present the redundancy rate with different transmission chances L.

we can see, Hairpin-lin (green line) does improve the trade-off compared to existing baselines (dashed blue line). Yet, there is still a half gap between Hairpin-lin and the Markov chain-based Hairpin (the red star). Therefore, it is necessary to analytically formulate the problem with the Markov chain to further push the trade-off forward.

Understanding Hairpin's decisions. We further present the redundancy rate results of Hairpin to provide a deeper understanding of how Hairpin optimizes in different scenarios. For redundancy rate, since the optimization of the absorbing Markov chain (§7.3.2) relies on the remaining transmission chance L, loss rate, remaining data packets to transmit,

and the frame size, we present the optimized redundancy rate over different parameters in Fig. D.8(a) to D.8(c). With more transmission chances, Hairpin would decrease the redundancy rate and rely on retransmissions for packet loss recovery. With fewer packets to retransmit, Hairpin also prefers a higher redundancy rate, as discussed in §7.3.2. Moreover, when the number of packets to transmit is small, the optimized redundancy rate is up to 500% in Fig. D.8(a), demonstrating the effectiveness of a redundancy rate of $>100\%$.

As for the optimization of block size, as we also discussed in §7.3.2, the optimal block size is the frame size (24 packets) in many cases. Nevertheless, as we discussed, at the decision boundary of remaining transmission chance, smaller block sizes do enjoy a slightly better performance by having additional transmission chances. As shown in Fig. D.8(d), although the optimal block size is the frame size in most cases, when the RTT is around 33ms and 50ms (the dividing point between 1, 2, and 3 transmission chances), the optimal block size might be smaller than the frame size. For example, compared to setting the block size to the frame size, the DMR with the optimized block size of Hairpin could be further reduced by $1.78\times$ around the RTT of 50ms and bottleneck bandwidth of 60Mbps. We optimize the block size for the last mile performance improvement.

E

Confucius (§8)

E.1 Fluid Model Analysis

In this section, we present the details about how we get the results in Table 8.2.

E.1.1 Fair Queueing (FQ)

Substituting Eq. 8.8a into Eq. 8.3, and taking the derivatives, we have:

$$\frac{\mathrm{d}^2}{\mathrm{d}t^2}s(t) + k \cdot s(t-\tau) = k\frac{C}{N+1} \tag{E.1}$$

With loss of generality, we assume $s(\tau) = C$, meaning that before N flows join, the sending rate has converged to the link capacity. Note that the measurement loop is usually much smaller than the control loop, i.e. $\tau \ll 1/k$, we then solve the differential equation above as:

$$s(t) = \left(1 - \frac{1}{N+1}\right)\cos\left(\sqrt{k}(t-\tau)\right) + \frac{1}{N+1}C \quad (t > \tau) \tag{E.2}$$

Since we are considering the transient conditions with a small t, where t is less than the first time of $s(t) = r(t)$, we approximate the formula above with Taylor's expression:

$$s(t) = C - C\frac{N}{N+1} \cdot \frac{k}{2} \cdot (t-\tau)^2 \quad (t > \tau) \tag{E.3}$$

Combine with Eq. 8.5, we have

$$q(t) = N\left(q_0 + \tau - \frac{N}{6k(N+1)}(t-\tau)^2\right) \tag{E.4}$$

We then have the maximum queue delay as:

$$q_{FQ}^{max} \geqslant q\left(\tau + \sqrt{2}k\right) = N\left(\frac{2}{3}\sqrt{\frac{2}{k}} + q_0 + \tau\right) \tag{E.5}$$

As N increases, q_{FIFO}^{max} will also increase.

Meanwhile, by substituting the available bandwidth in Eq. 8.7 with Eq. 8.8a, we have T_{FQ}:

$$T_{FQ} = \left(1 + \frac{1}{N}\right) \cdot \frac{NB}{C} \tag{E.6}$$

E.1.2 FIFO

Since the share of available bandwidth is proportional to the share of buffer occupancy, we estimate $r_{FIFO}(t)$ as in Eq. 8.8b. Similar to FQ, we can get:

$$q(t) \geqslant \frac{1}{C}\left(\frac{NB}{q_0 C}\right)\left(q_0 C + \int_0^t s(t')\mathrm{d}t' - tC\frac{1}{\frac{NB}{q_0 C}+1}\right) \tag{E.7}$$

and then

$$q_{FIFO}^{max} \geqslant q\left(\tau + \sqrt{\frac{2}{k}}\right) \tag{E.8}$$

Consequently

$$q_{FIFO}^{max} \geqslant \left(\frac{NB_0}{q_0 C}+1\right)\left(\frac{2}{3}\sqrt{\frac{2}{k}} + q_0 + \tau\right) \tag{E.9}$$

E.1.3 DRR

As we can see from Eq. 8.8c, the $r_{DRR}(t)$ is a special case of $r_{FQ}(t)$ with $N = 1$. Therefore, according to the delay degradation result in Eq. E.5, we have:

$$q_{DRR}^{max} \geqslant \frac{2}{3}\sqrt{\frac{2}{k}} + q_0 + \tau \tag{E.10}$$

The FCT satisfies:

$$T_{DRR} = \frac{2NB}{C} \tag{E.11}$$

In this case,

$$T_{DRR} - T_{FQ} = \frac{(N-1)B}{C}$$

diverges with N and B.

E.1.4 Confucius

For Confucius, we have:

$$r_{\text{Confucius}}(t) = \frac{C}{2} e^{-\lambda t} \quad (t > 0) \tag{E.12}$$

we could then solve out (using Laplacian transform, and solve with undetermined coefficients):

$$s(t) = Ae^{-\lambda(t-\tau)} + B\cos\sqrt{k}(t-\tau) \tag{E.13}$$

where

$$A = \quad C \cdot \tfrac{k}{2} \cdot \tfrac{1}{\lambda^2 + k \cdot e^{\lambda\tau}} \tag{E.14}$$

$$B = \qquad C - A \tag{E.15}$$

Still using Taylor's approximation:

$$\begin{aligned} s(t) \quad &= A\left(1 - \lambda(t-\tau)\right) + B\left(1 - \tfrac{1}{2}k(t-\tau)^2\right) \\ &= -\tfrac{B}{2}k(t-\tau)^2 - \lambda A(t-\tau) + A + B \end{aligned} \tag{E.16}$$

Denote the root of $s(t) = 0$ on $t > \tau$ as $t_0 + \tau$ $(t_0 > 0)$, we then have

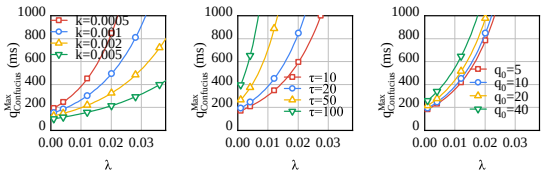
$$q(t_0 + \tau) = 2e^{\lambda(t_0+\tau)} \left(q_0 + \tau - \left(t_0 - \frac{\lambda A}{2C} t_0 - \frac{kB}{6C} t_0^3 \right) \right) \tag{E.17}$$

where t_0 satisfies:

$$t_0 = \frac{-\lambda A + \sqrt{(\lambda A)^2 + 2Bk(A+B)}}{Bk} \tag{E.18}$$

Thus, we have a bound of $q^{max}_{\text{Confucius}}$:

$$q^{max}_{\text{Confucius}} \approx q(t_0 + \tau) = f(\lambda; k, \tau, q_0) \tag{E.19}$$



(a) Changing k (b) Changing τ (c) Changing q_0

Figure E.1: The theoretical estimation from Confucius under different parameter settings.

independent of B or N. bounded. We expand the series as:

$$\begin{aligned} f(\lambda) &= F_0 + F_1\lambda + F_2\lambda^2 + o(\lambda^2) \\ F_0 &= 2q_0 + 6\tau + \frac{8}{2\sqrt{k}} \\ F_1 &= \frac{10}{3k} + 2q_0\tau + 2\tau^2 + \frac{4q_0}{\sqrt{k}} + \frac{16\tau}{3\sqrt{k}} \\ F_2 &= \frac{4q_0}{k} + \frac{6\tau}{k} + q_0\tau^2 + \tau^3 + \frac{6q_o\tau}{\sqrt{k}} + \frac{11\tau^2}{\sqrt{k}} \end{aligned} \tag{E.20}$$

Given that $\frac{1}{k} \ll q_0, \tau$, we can simplify and upper bound them into:

$$q^{max}_{\text{Confucius}} \leqslant 6q_0 + 15\tau + \frac{8\lambda}{k} + \frac{(10q_0 + 15\tau)\lambda^2}{k} \tag{E.21}$$

We further plot the unsimplified bound in different k and other parameter settings:

The FCT difference over the fair share for new flows is also bounded compared to other baselines. The FCT of N flows with B bytes, T for each flow basically follows:

Recall that $r(t) = \max(C - \frac{C}{2}2^{-\lambda t}, \frac{N}{N+1}C)$, we thus have

$$T_{\text{Confucius}} = \frac{(N+1)B}{C} + \frac{1}{\lambda}\cdot\left(\frac{1}{2} - \frac{1}{N}\log_2\frac{N+1}{2} - \frac{1}{2N}\right) \tag{E.22}$$

where $t \geqslant \frac{1}{\lambda}\log_2\frac{N+1}{2}$. In this case,

$$T_{\text{Confucius}} - T_{FQ} \leqslant \frac{1}{\lambda}\cdot\left(\frac{1}{2} - \frac{1}{N}\log_2\frac{N+1}{2} - \frac{1}{2N}\right) \leqslant \frac{\log_2 e}{\lambda} \tag{E.23}$$

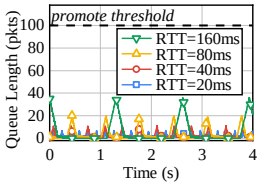


Figure E.2: The hysteresis design in Confucius (§8.5.2) is able to absorb the fluctuations caused by probing from CCAs.

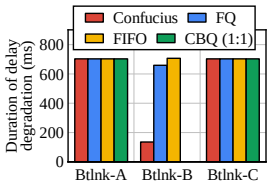


Figure E.3: When the bottleneck is elsewhere, Confucius maintains the same performance as existing mechanisms.

E.1.5 Responsiveness for CCAs

For different CCAs, we can fit their responsiveness k based on their probing period in the steady state. From the differential equations in Eq. 8.3 and Eq. 8.5, during the steady state where $r(t) \equiv C$, we can solve that the sending rate $s(t)$ follows:

$$s(t) = C + A\cos(\sqrt{k}t + \phi) \tag{E.24}$$

where A and ϕ are undetermined coefficients. In this case, we can know that the probing period of a CCA is $\frac{2\pi}{\sqrt{k}}$. From the respective design of CCAs, the probing period for Copa is 5 RTTs, and for BBR is 8 RTTs. For example, when RTT is 40 ms, we will have $k_{Copa} = 0.001\ (ms^{-2})$, $k_{BBR} = 0.0004\ (ms^{-2})$.

E.2 Supplementary Experiments

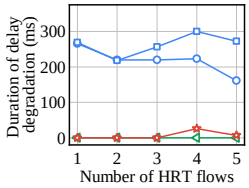
We further evaluate the performance of Confucius in a series of microbenchmarking settings.

E.2.1 Working with Bandwidth Probing

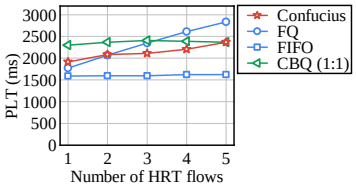
Some recent CCAs proposed to periodically probe the available bandwidth by overshooting the network, which might introduce noises in classifying the buffer occupancy of flows in Confucius. Some recent examples for video streaming include Sprout [261], PCC (probing up to 5%) [99], and BBR (probing 25%) [75]. We evaluate how Confucius is able to handle the bandwidth probing from CCAs. We first run one BBR flow, which is the most aggressive one among these bandwidth probing CCAs, and change the RTT from 20 ms to 160 ms since the probing period is counted in the unit of RTT. As shown in Figure E.2, with the other settings the same as Figure 8.8, the queue fluctuations never go across the threshold of reclassification of the flow. This is due to the hysteresis design in §8.5.2 – Confucius deliberately makes conservative decisions in the classification of flows to smoothize the noises out. This can also be validated from Figure 8.11(b): the classification results are stable all the time even if BBR periodically probes the bandwidth. Therefore, Confucius is able to work well with bandwidth-probing CCAs.

E.2.2 Working with Different Bottleneck

We further evaluate the end-to-end performance when the bottleneck is not where Confucius is deployed. Confucius is able to reduce the latency volatility when it is deployed on the bottleneck router. Our further experiments show that Confucius does not introduce side effects when the bottleneck is before or after the router deployed with Confucius. We still deploy queue management mechanisms to the router before link B and respectively rate-limit the link A, B, and C in Figure 8.8 to 20 Mbps:



(a) Duration of delay degradation of the HRT (old) flow.



(b) PLT of Web (new) flows.

Figure E.4: We increase the number of simultaneous HRT flows, and measure the results again with the Alexa dataset.

- `Btlnk-A`. When link A is limited while the other two links are set to 100 Mbps, the bottleneck is before the place of Confucius.
- `Btlnk-B`. The case when link B is limited is what we mainly evaluated in this section, where Confucius is at the bottleneck.
- `Btlnk-C`. When link C is limited, the bottleneck is after the place of Confucius.

For those unmanaged routers, they adopt FIFO as their default mechanism. As shown in Figure E.3, the performance is only affected by the mechanism deployed at the bottleneck. When Confucius is not at the bottleneck (e.g., link A or C), the performance is the same no matter what mechanism is deployed at link B. It is worth to note that as discussed in a series of papers [56, 188], the last-mile routers (e.g., cellular base stations, home wireless APs) are the bottleneck for most of the congestions, in which case deploying Confucius will achieve significant performance benefits.

E.2.3 Multiple HRT Flows Competition

We further evaluate the performance when there are multiple HRT flows running simultaneously. We reproduce the experiments in Figure 8.9(a) but change the number of HRT flows from 1 to 5. The average duration of delay degradation of HRT flows, and the PLT of Web flows are presented in Figure E.4. Confucius is able to provide a consistent performance for multiple HRT flows in the same time – the delay degradation is consistently negligible independent of the number of concurrent HRT flows and the PLT stays roughly the same place compared to the baselines. Note that since Confucius is designed for last-mile routers, 5 concurrent flows should be able to cover most scenarios [188].

References

[1] Har (file format) - wikipedia. https://en.wikipedia.org/wiki/HAR_(file_format).

[2] Pareto front - wikipedia. https://en.wikipedia.org/wiki/Pareto_front.

[3] Tcp analysis | cs-224-lectures. https://grubdragon.github.io/CS-224-Lectures/lec/lec11.html.

[4] x264 - wikipedia. https://en.wikipedia.org/wiki/X264, .

[5] x265 - wikipedia. https://en.wikipedia.org/wiki/X265, .

[6] [openwrt wiki] netgear wndr3800. https://openwrt.org/toh/netgear/wndr3800, 2011.

[7] [openwrt wiki] tp-link tl-wdr4900. https://openwrt.org/toh/tp-link/tl-wdr4900, 2013.

[8] [systemd-devel] [announce] systemd 217. https://lists.freedesktop.org/archives/systemd-devel/2014-October/024662.html, 2014.

[9] Raw data - measuring broadband america. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-2016, 2016.

[10] 5g can make remote driving a reality, telefónica and ericsson demostrate at mwc. https://www.telefonica.com/en/web/press-office/-/5g-can-make-remote-driving-a-reality-telefonica-and-ericsson-demostrate-at-mwc, 2017.

[11] Dash.js. https://github.com/Dash-Industry-Forum/dash.js, 2018.

[12] Your games. your devices. play anywhere | nvidia geforce now. https://www.nvidia.com/en-us/geforce-now/, 2020.

[13] Critical services report: Video conferencing (uk) | blog. https://samknows.com/blog/critical-services-report-video-conferencing-uk, 2020.

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

[15] Stadia - one place for all the ways we play. https://stadia.google.com/, 2020.

[16] Start - tencent cloud gaming. https://start.qq.com/, 2020.

[17] Issue 93006: Update to media_opt_util: - code review. https://webrtc-codereview.appspot.com/93006, 2020.

[18] Psa: Webrtc m88 release notes. https://groups.google.com/g/discuss-webrtc/c/A0FjOcTW2c0/m/UAv-veyPCAAJ, 2020.

[19] Cloud gaming (beta) with xbox game pass | xbox. https://www.xbox.com/en-US/xbox-game-pass/cloud-gaming, 2020.

[20] Facebook 360 video. https://facebook360.fb.com/, 2021.

[21] Finite element method - wikipedia. https://en.wikipedia.org/wiki/Finite_element_method, 2021.

[22] Trtx 2080 ti vs rtx 3080 ti game performance benchmarks (i7-8700k vs core i9-10900k) - gpucheck united states / usa. https://www.gpucheck.com/compare/nvidia-geforce-rtx-2080-ti-vs-nvidia-geforce-rtx-3080-ti/, 2021.

[23] Google meet and default video resolution - google meet community. https://support.google.com/meet/thread/58039897/google-meet-and-default-video-resolution, 2021.

[24] Huawei video conferencing platform — huawei enterprise. https://e.huawei.com/en/solutions/enterprise-collaboration/videoconferencing-platform, 2021.

[25] Vr-interactive – we are interactive. https://vr-interactive.at/, 2021.

[26] Prepare your network for meet video calls - google workspace admin help. https://support.google.com/a/answer/1279090, 2021.

[27] Optimizing 5g for a new class of low-latency experiences [video]. https://www.qualcomm.com/news/onq/2021/07/20/optimizing-5g-new-class-low-latency-experiences, 2021.

[28] Manwuyixiang roast lamb leg □□□□□□□□·□□□ (□□□□)□. http://cnc.www.dianping.com/shop/igEL946mgXy0B2KV, 2021.

[29] Troubleshooting your stadia experience - stadia help. https://support.google.com/stadia/answer/9595943, 2021.

[30] Meeting and phone statistics – zoom help center. https://support.zoom.us/hc/en-us/articles/202920719-Meeting-and-phone-statistics, 2021.

[31] Webrtc samples. https://webrtc.github.io/samples/, 2021.

[32] Youtube vr - home - youtube vr. https://vr.youtube.com/, 2021.

[33] 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, 2021.

[34] Alexa Top Websites >> ExpiredDomains.net. https://member.expireddomains.net/domains/researchalexamillion/, 2022.

[35] Cisco vni complete forecast highlights global - consumer highlights. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_Device_Growth_Traffic_Profiles.pdf, 2022.

[36] multithreading - pin processor cpu isolation on windows - stack overflow. https://stackoverflow.com/questions/15324586/pin-processor-cpu-isolation-on-windows, 2022.

[37] Processor benchmarks - geekbench browser. https://browser.geekbench.com/processor-benchmarks/, 2022.

[38] Gfxbench - unified graphics benchmark based on dxbenchmark (directx) and glbenchmark (opengl es). https://gfxbench.com/result.jsp, 2022.

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

[40] Vamsi Addanki, Maria Apostolaki, Manya Ghobadi, Stefan Schmid, and Laurent Vanbever. Abm: Active buffer management in datacenters. In *Proc. ACM SIGCOMM*, 2022.

[41] Mohammad Alizadeh, Abdul Kabbani, Berk Atikoglu, and Balaji Prabhakar. Stability analysis of qcn: the averaging principle. In *Proc. ACM SIGMETRICS*, 2011.

[42] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: Minimal near-optimal datacenter transport. In *Proc. ACM SIGCOMM*, 2013.

[43] Amit. Huawei news | huawei launched cloud mobile phone. https://www.huaweiupdate.com/huawei-launched-cloud-mobile-phone/, 2020.

[44] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing router buffers. *ACM SIGCOMM Computer Communication Review*, 34(4):281–292, 2004.

[45] ArsTechnica. Nvidia gtx 1080 review: The new performance king. https://arstechnica.com/gadgets/2016/05/nvidia-gtx-1080-review/4/, 2016.

[46] Venkat Arun. Implementation of the copa congestion control algorithm using ccp. https://github.com/venkatarun95/ccp_copa, 2020.

[47] Venkat Arun and Hari Balakrishnan. Copa: Practical delay-based congestion control for the internet. In *Proc. USENIX NSDI*, 2018.

[48] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. Starvation in End-to-End Congestion Control. In *Proc. ACM SIGCOMM*, 2022.

[49] Salahuddin Azad, Wei Song, and Dian Tjondronegoro. Bitrate modeling of scalable videos using quantization parameter, frame rate and spatial resolution. In *Proc. IEEE ICASSP*, pages 2334–2337, 2010.

[50] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Proc. NIPS*, 2014.

[51] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *Proc. USENIX NSDI*, 2015.

[52] Vaibhav Bajpai, Steffie Jacob Eravuchira, and Jürgen Schönwälder. Dissecting last-mile latency characteristics. *ACM SIGCOMM Computer Communication Review*, 47(5):25–34, 2017.

[53] Fred Baker, Jozef Babiarz, and Kwok Ho Chan. Configuration Guidelines for DiffServ Service Classes. IETF RFC 4594, 2006.

[54] Luca Baldantoni, Henrik Lundqvist, and Gunnar Karlsson. Adaptive end-to-end fec for improving tcp performance over wireless links. In *Proc. IEEE ICC*, 2004.

[55] Matthew Ball and Jacob Navok. Challenge #3: Enormous bandwidth costs and operational burdens | cloud gaming: Why it matters and the games it will create. https://www.matthewball.vc/all/cloudmiles, 2020.

[56] Nimantha Baranasuriya, Vishnu Navda, Venkata N Padmanabhan, and Seth Gilbert. Qprobe: Locating the bottleneck in cellular communication. In *Proc. ACM CoNEXT*, pages 1–7, 2015.

[57] Asha Barbaschow. Alibaba unveils cloud 2.0, wuying cloud computer, and xiaomanlv logistics robot. https://www.zdnet.com/article/alibaba-unveils-cloud-2-0-wuying-cloud-computer-and-xiaomanlv-logistics-robot/, 2020.

[58] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. In *Proc. NeurIPS*, 2018.

[59] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. Network dissection: Quantifying interpretability of deep visual representations. In *Proc. IEEE CVPR*, 2017.

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

[61] Richard Bellman and Robert Kalaba. On adaptive control processes. *IRE Transactions on Automatic Control*, 4(2):1–9, 1959.

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

[63] Ankita Bhutani and Preeti Wadhwani. Cloud gaming market share forecast 2025 | industry size report. https://www.gminsights.com/industry-analysis/cloud-gaming-market, 2020.

[64] Ethan Blanton, Dr. Vern Paxson, and Mark Allman. TCP Congestion Control. IETF RFC 5681, 2009.

[65] Hendrik Blockeel and Luc De Raedt. Top-down induction of first-order logical decision trees. *Artificial intelligence*, 101(1-2):285–297, 1998.

[66] J-C Bolot, Sacha Fosse-Parisis, and Don Towsley. Adaptive fec-based error control for internet telephony. In *Proc. IEEE INFOCOM*, 1999.

[67] brianhu. 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, 2021.

[68] Karl Bridge and Michael Satran. Multitasking - win32 apps | microsoft docs. https://docs.microsoft.com/en-us/windows/win32/procthread/multitasking, 2018.

[69] Bob Briscoe, Koen De Schepper, Marcelo Bagnulo, and Greg White. Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture. RFC 9330, January 2023. URL https://www.rfc-editor.org/info/rfc9330.

[70] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint 2005.14165*, 2020.

[71] James Bruce, Marta Mrak, and Rajitha Weerakkody. Testing av1 and vvc - bbc r&d. https://www.bbc.co.uk/rd/blog/2019-05-av1-codec-streaming-processing-hevc-vvc, 2019.

[72] Alan Bryman and Duncan Cramer. *Quantitative data analysis with IBM SPSS 17, 18 & 19: A guide for social scientists*. Routledge, 2012.

[73] James Bulman and Peter Garraghan. A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines. In *Proc. USENIX HotCloud*, 2020.

[74] Ronald S. Bultje. The world's fastest vp9 decoder: ffvp9. https://blogs.gnome.org/rbultje/2014/02/22/the-worlds-fastest-vp9-decoder-ffvp9/, 2014.

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

[76] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of ACM International Conference on Multimedia Systems (MMSys)*, 2016.

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

[78] Marc Carrascosa and Boris Bellalta. Cloud-gaming: Analysis of google stadia traffic. *arXiv:2009.09786*, 2020.

[79] Gwyn Chatranon, Miguel A Labrador, and Sujata Banerjee. Black: detection and preferential dropping of high bandwidth unresponsive flows. In *Proc. IEEE ICC*, 2003.

[80] Jianhui Chen, Hoang M Le, Peter Carr, Yisong Yue, and James J Little. Learning online smooth predictors for realtime camera planning using recurrent decision trees. In *Proc. IEEE CVPR*, 2016.

[81] Ke Chen, Han Wang, Shuwen Fang, Xiaotian Li, Minghao Ye, and H. Jonathan Chao. Rl-afec: Adaptive forward error correction for real-time video communication based on reinforcement learning. In *Proc. ACM MMSys*, 2022.

[82] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. Scheduling mix-flows in commodity datacenters with karuna. In *Proc. ACM SIGCOMM*, 2016.

[83] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proc. ACM SIGCOMM*, 2018.

[84] Wei Chen, Liangping Ma, and Chien-Chung Shen. Congestion-aware mac layer adaptation to improve video teleconferencing over wi-fi. In *Proceedings of ACM Multimedia Systems Conference (MMSys)*, 2015.

[85] Sheng Cheng, Han Hu, Xinggong Zhang, and Zongming Guo. Rebuffering but not suffering: Exploring continuous-time quantitative qoe by user's exiting behaviors. In *Proc. IEEE INFOCOM*, 2023.

[86] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. The RACK-TLP Loss Detection Algorithm for TCP. IETF RFC 8985, 2021.

[87] Yushin Cho, William A Pearlman, and Amir Said. Low complexity resolution progressive image coding algorithm: progres (progressive resolution decompression). In *Proc. IEEE ICIP*, 2005.

[88] Tzu-Der Chuang, Pei-Kuei Tsung, Pin-Chih Lin, Lo-Mei Chang, Tsung-Chuan Ma, Yi-Hau Chen, and Liang-Gee Chen. A 59.5 mw scalable/multi-view video decoder chip for quad/3d full hdtv and video streaming applications. In *Proc. IEEE ISSCC*, pages 330–331, 2010.

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

[90] Harald Cramér. Mathematical methods of statistics, 1946. *Department of Mathematical SU*, 1946.

[91] Yousri Daldoul, Djamal-Eddine Meddour, and Adlen Ksentini. Performance evaluation of ofdma and mu-mimo in 802.11 ax networks. *Computer Networks*, 2020.

[92] Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *Proc. USENIX NSDI*, 2022.

[93] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. IEEE CVPR*, 2009.

[94] Arnaud Dethise, Marco Canini, and Srikanth Kandula. Cracking open the black box: What observations can tell us about reinforcement learning agents. In *Proc. ACM NetAI*, 2019.

[95] Amogh Dhamdhere, David D Clark, Alexander Gamero-Garrido, Matthew Luckie, Ricky KP Mok, Gautam Akiwate, Kabir Gogia, Vaibhav Bajpai, Alex C Snoeren, and Kc Claffy. Inferring persistent interdomain congestion. In *Proc. ACM SIGCOMM*, pages 1–15, 2018.

[96] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. A network analysis on cloud gaming: Stadia, geforce now and psnow. *Network*, 2021.

[97] Robert G. Gallager Dimitri P. Bertsekas. Section 3.3: The m/m/1 queuing system. In *Data Networks (2nd Edition)*, 1992.

[98] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. In *Proc. ACM SIGCOMM*, 2011.

[99] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. Pcc: Re-architecting congestion control for consistent high performance. In *Proc. USENIX NSDI*, 2015.

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

[101] Mengnan Du, Ninghao Liu, and Xia Hu. Techniques for interpretable machine learning. *Commun. ACM*, pages 68–77, 2020.

[102] Nandita Dukkipati, Tiziana Refice, Yuchung Cheng, Jerry Chu, Tom Herbert, Amit Agarwal, Arvind Jain, and Natalia Sutin. An argument for increasing tcp's initial congestion window. *ACM SIGCOMM Computer Communication Review*, pages 26–33, 2010.

[103] Pierre Ecarlat. Cnn - do we need to go deeper? https://medium.com/finc-engineering/cnn-do-we-need-to-go-deeper-afe1041e263e, 2017.

[104] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 2019.

[105] Cristian Estan and George Varghese. New directions in traffic measurement and accounting. In *Proc. ACM SIGCOMM*, pages 323–336, 2002.

[106] Theodore Faber. Acc: using active networking to enhance feedback congestion control mechanisms. *IEEE network*, 1998.

[107] Ferenc Fejes, Gergő Gombos, Sándor Laki, and Szilveszter Nádas. Who will save the internet from the congestion control revolution? In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 1–6, 2019.

[108] Wu-chang Feng, Dilip Kandlur, Debanjan Saha, and Kang Shin. Blue: A new class of active queue management algorithms. 1999.

[109] Wu-chang Feng, Dilip D Kandlur, Debanjan Saha, and Kang G Shin. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *Proc. IEEE INFOCOM*, 2001.

[110] Wu-chun Feng, Apu Kapadia, and Sunil Thulasidasan. Green: proactive queue management over a best-effort network. In *Proc. IEEE GLOBECOM*, 2002.

[111] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *Proc. ACM SIGCOMM*, 2013.

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

[113] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on networking*, 1993.

[114] Mary Jo Foley. Microsoft marches toward launching its 'cloud pc' service, possibly this summer. https://www.zdnet.com/article/microsoft-marches-toward-launching-its-cloud-pc-service-possibly-this-summer/, 2021.

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

[116] Silas L Fong, Ashish Khisti, Baochun Li, Wai-Tian Tan, Xiaoqing Zhu, and John Apostolopoulos. Optimal streaming codes for channels with burst and arbitrary erasures. *IEEE Transactions on Information Theory*, 2019.

[117] Romain Fontugne, Anant Shah, and Kenjiro Cho. Persistent last-mile congestion: not so uncommon. In *Proc. ACM IMC*, pages 420–427, 2020.

[118] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. USENIX NSDI*, 2017.

[119] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *Proc. USENIX NSDI*, 2018.

[120] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *Proc. ICLR*, 2019.

[121] Jerome H Friedman, Richard A Olshen, Charles J Stone, et al. Classification and regression trees. *Wadsworth & Brooks*, 1984.

[122] Nitin Garg. Evaluating copa congestion control for improved video performance. https://engineering.fb.com/2019/11/17/video-engineering/copa/, 2019.

[123] GFXBench. 3d graphics performance of google pixel c. https://gfxbench.com/device.jsp?D=Google+Pixel+C, 2017.

[124] Moinak Ghoshal, Pranab Dash, Zhaoning Kong, Qian Xu, Y.Charlie Hu, Dimitrios Koutsonikolas, and Yuanjie Li. Can 5g mmwave enable multi-user ar apps? In *Proc. PAM*, 2022.

[125] Prateesh Goyal, Anup Agarwal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. Abc: A simple explicit congestion controller for wireless networks. In *Proc. USENIX NSDI*, 2020.

[126] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proc. ACM SIGCOMM*. 2019.

[127] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. A survey of methods for explaining black box models. *ACM Computing Surveys (CSUR)*, 2018.

[128] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. Lemna: Explaining deep learning based security applications. In *Proc. ACM CCS*, 2018.

[129] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 2008.

[130] Jefferson Han and Brian Smith. Cu-seeme vr immersive desktop teleconferencing. In *Proc. ACM Multimedia*, pages 199–207, 1997.

[131] Boris Hanin and David Rolnick. Complexity of linear regions in deep networks. In *Proc. ICML*, 2019.

[132] Osama Haq, Mamoon Raja, and Fahad R Dogar. Measuring and improving the reliability of wide-area cloud paths. In *Proc. WWW*, 2017.

[133] Refael Hassin and Moshe Haviv. *To queue or not to queue: Equilibrium behavior in queueing systems*, volume 59. Springer Science & Business Media, 2003.

[134] Juyeon Heo, Sunghwan Joo, and Taesup Moon. Fooling neural network interpretations via adversarial model manipulation. In *Proc. NeurIPS*, 2019.

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

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

[137] Stefan Holmer, Mikhal Shemer, and Marco Paniconi. Handling packet loss in webrtc. In *2013 IEEE International Conference on Image Processing*, 2013.

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

[139] Petr Holub, Jiří Matela, Martin Pulec, and Martin Šrom. Ultragrid: low-latency high-quality video transmissions on commodity hardware. In *Proc. ACM International Conference on Multimedia*, 2012.

[140] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. Gaminganywhere: an open cloud gaming system. In *Proc. ACM Multimedia Systems Conference (MMSys)*, 2013.

[141] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proc. ACM SIGCOMM*, 2014.

[142] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proc. ACM MobiSys*, 2017.

[143] Zenja Ivkovic, Ian Stavness, Carl Gutwin, and Steven Sutcliffe. Quantifying and mitigating the negative effects of local latencies on aiming in 3d shooter games. In *Proc. ACM CHI*, pages 135–144, 2015.

[144] Jana Iyengar and Ian Swett. Quic loss detection and congestion control. *IETF RFC 9002*, 2021.

[145] Van Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM*, 1988.

[146] Manish Jain and Constantinos Dovrolis. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput. In *Proc. ACM SIGCOMM*, 2002.

[147] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *Proc. ICML*, 2019.

[148] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proc. ACM CoNEXT*, 2012.

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

[150] Alan Jones, Peter Sevcik, and Rebecca Wetzel. 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, 2021.

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

[152] Ad Kamerman and Leo Monteban. Wavelan®-ii: a high-performance wireless lan for the unlicensed band. *Bell Labs technical journal*, 1997.

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

[154] JFC Kingman and MF Atiyah. The single server queue in heavy traffic. *Oper. Manage., Critical Perspect. Bus. Manage*, 2003.

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

[156] Ingo Kofler, Martin Prangl, Robert Kuschnig, and Hermann Hellwagner. An h. 264/svc-based adaptation proxy on a wifi router. In *Proc. NOSSDAV*, 2008.

[157] M Nikhil Krishnan, Deeptanshu Shukla, and P Vijay Kumar. Rate-optimal streaming codes for channels with burst and random erasures. *IEEE Transactions on Information Theory*, 2020.

[158] Marwan Krunz and Herman Hughes. A traffic for mpeg-coded vbr streams. In *Proc. ACM SIGMETRICS*, 1995.

[159] Ana Kuzmanic and Vlasta Zanchi. Hand shape classification using dtw and lcss as similarity measures for vision-based gesture recognition system. In *EUROCON 2007-The International Conference on" Computer as a Tool"*, pages 264–269. IEEE, 2007.

[160] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436, 2015.

[161] Yun Gu Lee and Byung Cheol Song. An intra-frame rate control algorithm for ultralow delay h. 264/advanced video coding (avc). *IEEE Transactions on Circuits and Systems for Video Technology*, pages 747–752, 2009.

[162] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

[163] Tong Li, Kai Zheng, Ke Xu, Rahul Arvind Jadhav, Tao Xiong, Keith Winstein, and Kun Tan. Tack: Improving wireless transport performance by taming acknowledgments. In *Proc. ACM SIGCOMM*, 2020.

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

[165] Zhi Li, Xiaoqing Zhu, Joshua Gahm, Rong Pan, Hao Hu, Ali C Begen, and David Oran. Probe and adapt: Rate adaptation for http video streaming at scale. *IEEE J. Sel. Areas Commun.*, pages 719–733, 2014.

[166] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, and Megha Manohara. Toward a practical perceptual video quality metric | netflix techblog. https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652, 2016.

[167] CC Lin, JI Guo, HC Chang, YC Yang, JW Chen, MC Tsai, and JS Wang. A 160kgate 4.5 kb skram h. 264 video decoder for hdtv applications. In *Proc. IEEE ISSCC*, pages 1596–1605, 2006.

[168] Dong Lin and Robert Morris. Dynamics of random early detection. In *Proc. ACM SIGCOMM*, 1997.

[169] Candice Liu. Hardware decoding vs software decoding in 4k h264/h265 video. https://www.macxdvd.com/mac-video-converter-pro/hardware-decoding-4k-ultra-hd-video.htm, 2020.

[170] Ruilin Liu, Daehan Kwak, Srinivas Devarakonda, Kostas Bekris, and Liviu Iftode. Investigating remote driving over the lte network. In *Proceedings of the 9th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, pages 264–269, 2017.

[171] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the transience-equilibrium nexus: A new approach to datacenter packet transport. In *Proc. USENIX NSDI*, 2021.

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

[173] Chengnian Long, Bin Zhao, Xinping Guan, and Jun Yang. The yellow active queue management algorithm. *Elsevier Computer Networks*, 2005.

[174] Andrea Lottarini, Alex Ramirez, Joel Coburn, Martha A Kim, Parthasarathy Ranganathan, Daniel Stodolsky, and Mark Wachsler. vbench: Benchmarking video transcoding in the cloud. In *Proc. ASPLOS*, pages 797–809, 2018.

[175] James M Lucas and Michael S Saccucci. Exponentially weighted moving average control schemes: properties and enhancements. *Technometrics*, 32(1):1–12, 1990.

[176] Mike H MacGregor and Weiguang Shi. Deficits for bursty latency-critical flows: DRR++. In *Proc. IEEE International Conference on Networks (ICON)*, pages 287–293.

[177] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of Berkeley symposium on mathematical statistics and probability*, 1967.

[178] Chaitanya Manapragada, Geoffrey I Webb, and Mahsa Salehi. Extremely fast decision tree. In *Proc. ACM KDD*, 2018.

[179] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proc. ACM SIGCOMM*, 2017.

[180] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. In *ICML Reinforcement Learning for Real Life Workshop*, 2019.

[181] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proc. ACM SIGCOMM*, 2019.

[182] Bill Marczak and John Scott-Railton. 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/, 2020.

[183] Gustavo Marfia, Claudio E Palazzi, Giovanni Pau, Mario Gerla, and Marco Roccetti. Tcp libra: Derivation, analysis, and comparison with other rtt-fair tcps. *Computer Networks*, 2010.

[184] Zili Meng and Mingwei Xu. Latency optimization in real-time multimedia transport: Architecture, progress and the future (in chinese). *Journal of Computer Research and Development*, 2023.

[185] Zili Meng, Jing Chen, Yaning Guo, Chen Sun, Hongxin Hu, and Mingwei Xu. Pitree: Practical implementation of abr algorithms using decision trees. In *Proc. ACM MM*, 2019.

[186] Zili Meng, Minhu Wang, Jiasong Bai, Mingwei Xu, Hongzi Mao, and Hongxin Hu. Interpreting deep learning-based networking systems. In *Proc. ACM SIGCOMM*, 2020.

[187] Zili Meng, Yaning Guo, Yixin Shen, Jing Chen, Chao Zhou, Minhu Wang, Jia Zhang, Mingwei Xu, Chen Sun, and Hongxin Hu. Practically deploying heavyweight adaptive bitrate algorithms with teacher-student learning. *IEEE/ACM Transactions on Networking*, 2021.

[188] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving Consistent Low Latency for Wireless Real Time Communications with the Shortest Control Loop. In *Proc. ACM SIGCOMM*, 2022.

[189] Zili Meng, Nirav Atre, Mingwei Xu, Justine Sherry, and Maria Apostolaki. Confucius queue management: Be fair but not too fast. arXiv preprint 2310.18030, 2023.

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

[191] Zili Meng, Xiao Kong, Jing Chen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Hairpin: Rethinking packet loss recovery in edge-based interactive video streaming. In *Proc. USENIX NSDI*, 2024.

[192] John Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine learning*, 4(2):227–243, 1989.

[193] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The great internet tcp congestion control census. In *Proc. ACM Sigmetrics*, 2020.

[194] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal packet scheduling. In *Proc. USENIX NSDI*, 2016.

[195] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, 2013.

[196] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[197] China Mobile and ZTE. Powered by sa: 5g mec-based cloud game innovation practice. GSMA 5G Case Studies (https://www.gsma.com/futurenetworks/wp-content/uploads/2020/03/Powered-by-SA-5G-MEC-Based-Cloud-Game-Innovation-Practice-.pdf), 2020.

[198] Chirag Modi, Dhiren Patel, Bhavesh Borisaniya, Hiren Patel, Avi Patel, and Muttukrishnan Rajarajan. A survey of intrusion detection techniques in cloud. *Elsevier Journal of network and computer applications*, pages 42–57, 2013.

[199] Nitinder Mohan, Lorenzo Corneo, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, and Jussi Kangasharju. Pruning edge research with latency shears. In *Proc. ACM HotNets*, 2020.

[200] Omar Mossad, Khaled Diab, Ihab Amer, and Mohamed Hefeeda. Deepgame: Efficient video encoding for cloud gaming. In *Proc. ACM Multimedia*, 2021.

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

[202] Nina Narodytska, Alexey Ignatiev, Filipe Pereira, Joao Marques-Silva, and IS RAS. Learning optimal decision trees with sat. In *Proc. IJCAI*, 2018.

[203] Kathleen Nichols and Van Jacobson. Controlling queue delay. *Communications of the ACM*, 2012.

[204] Vit Niennattrakul and Chotirat Ann Ratanamahatana. On clustering multimedia time series data using k-means and dynamic time warping. In *2007 International Conference on Multimedia and Ubiquitous Engineering (MUE'07)*, pages 733–738. IEEE, 2007.

[205] Ilya Nikolaevskiy. Refactor framebuffer to store decoded frames history separately (i82beoeb3) · gerrit code review. https://webrtc-review.googlesource.com/c/src/+/116686, 2019.

[206] Roman Novak, Yasaman Bahri, Daniel A Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: an empirical study. In *Proc. ICLR*, 2018.

[207] OPG609. List of 60fps games playable on ps5. https://www.reddit.com/r/PS5/comments/kiuh2t/list_of_60fps_games_playable_on_ps5/, 2020.

[208] Chinmay Padhye, Kenneth J Christensen, and Wilfrido Moreno. A new adaptive fec loss control algorithm for voice over ip applications. In *Proc. IEEE INFOCOM*, 2000.

[209] Jitendra Padhye, Victor Firoiu, Donald F Towsley, and James F Kurose. Modeling tcp reno performance: a simple model and its empirical validation. *IEEE/ACM transactions on Networking*, 8(2):133–145, 2000.

[210] Rong Pan, Lee Breslau, Balaji Prabhakar, and Scott Shenker. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review*, 33(2):23–39, 2003.

[211] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[212] Adrian Pennington. So you say you're planning a 16k live stream... - nab amplify. https://amplify.nabshow.com/articles/so-you-say-youre-planning-a-16k-live-stream/, 2022.

[213] Stefano Petrangeli, Viswanathan Swaminathan, Mohammad Hosseini, and Filip De Turck. An http/2-based adaptive streaming framework for 360 virtual reality videos. In *Proc. ACM Multimedia*, 2017.

[214] Alok Prakash, Hussam Amrouch, Muhammad Shafique, Tulika Mitra, and Jörg Henkel. Improving mobile gaming performance through cooperative cpu-gpu thermal management. In *Proc. ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2016.

[215] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 1994.

[216] ITU Recommendations. G.1070 : Opinion model for video-telephony applications. https://www.itu.int/rec/T-REC-G.1070, 2018.

[217] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proc. ACM KDD*, 2016.

[218] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. Semantically equivalent adversarial rules for debugging nlp models. In *Proc. ACL*, 2018.

[219] Haakon Riiser, Paul Vigmostad, Carsten Griwodz, and Pål Halvorsen. Commute path bandwidth traces from 3g networks: Analysis and applications. In *Proc. ACM MMSys*, 2013.

[220] Vincent Roca, Jani Peltotalo, Jerome Lacan, and Sami Peltotalo. Reed-Solomon Forward Error Correction (FEC) Schemes. IETF RFC 5510, 2009.

[221] Vincent Roca, Mathieu Cunche, Jerome Lacan, Amine Bouabdallah, and Kazuhisa Matsuzono. Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME. IETF RFC 6865, 2013.

[222] Elizabeth Ross, John Parente, Mike Jacobs, David Kuehn, John Baldwin, Corey Plett, Brock Mammen, and Liza Poggemeyer. typeperf | microsoft docs. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/typeperf, 2017.

[223] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proc. AISTATS*, 2011.

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

[225] Michael Rudow, Francis Y. Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and K.V. Rashmi. Streammelt: Efficient loss recovery for videoconferencing via streaming codes. In *Proc. USENIX NSDI*, 2023.

[226] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Unveiling the potential of graph neural networks for network modeling and optimization in sdn. In *Proc. ACM SOSR*, 2019.

[227] Saeed Shafiee Sabet, Steven Schmidt, Saman Zadtootaghaj, Babak Naderi, Carsten Griwodz, and Sebastian Möller. A latency compensation technique based on game characteristics to mitigate the influence of delay on cloud gaming quality of experience. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 15–25, 2020.

[228] Matt Sargent, Jerry Chu, Dr. Vern Paxson, and Mark Allman. Computing TCP's Retransmission Timer. IETF RFC 6298.

[229] Zaheduzzaman Sarker, Colin Perkins, Varun Singh, and M Ramalho. Rtp control protocol (rtcp) feedback for congestion control. *IETF RFC 8888*, 2021.

[230] Pasi Sarolahti, Markku Kojo, and Kimmo Raatikainen. F-rto: an enhanced recovery algorithm for tcp retransmission timeouts. *ACM SIGCOMM Computer Communication Review*, 2003.

[231] Koen De Schepper, Bob Briscoe, and Greg White. Dual-Queue Coupled Active Queue Management (AQM) for Low Latency, Low Loss, and Scalable Throughput (L4S). RFC 9332, January 2023. URL https://www.rfc-editor.org/info/rfc9332.

[232] H Schulzrinne, A Rao, R Lanphier, M Westerlund, and M Stiemerling. Real-time streaming protocol version 2.0. IETF RFC 7826, 2016.

[233] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, and Van Jacobson. RTP: A Transport Protocol for Real-Time Applications. IETF RFC 3550, 2003.

[234] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, 2007.

[235] Satadal Sengupta, Niloy Ganguly, Sandip Chakraborty, and Pradipta De. Hotdash: Hotspot aware adaptive video streaming using deep reinforcement learning. In *Proc. IEEE ICNP*, pages 165–175, 2018.

[236] Arun Kumar Sharma. *Text book of correlations and regression*. Discovery Publishing House, 2005.

[237] Yueshi Shen. 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/, 2017.

[238] Hang Shi, Yong Cui, Feng Qian, and Yuming Hu. Dtp: Deadline-aware transport protocol. In *Proc. APNet*, 2019.

[239] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proc. ACM Multimedia*, 2011.

[240] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proc. ACM SIGCOMM*, 1995.

[241] Ivan Slivar, Lea Skorin-Kapov, and Mirko Suznjevic. Cloud gaming qoe models for deriving video encoding adaptation strategies. In *Proc. ACM Multimedia Systems Conference (MMSys)*, 2016.

[242] Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley, Rajat Monga, Greg Corrado, Fernanda B. Viégas, and Martin Wattenberg. Tensorflow. js: Machine learning for the web and beyond. In *Proc. SysML*, 2019.

[243] Kaarmukilan S.P. What is hairpin net shot in badminton? - quora. https://www.quora.com/What-is-hairpin-net-shot-in-badminton/answer/Kaarmukilan-S-P, 2020.

[244] Bruce Spang, Serhat Arslan, and Nick McKeown. Updating the theory of buffer sizing. *Performance Evaluation*, 151:102232, 2021.

[245] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. In *Proc. IEEE INFOCOM*, 2016.

[246] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: improving bitrate adaptation in the dash reference player. In *Proc. ACM MMSys*, 2018.

[247] James Stringer. Pushing it to the limit – parsec at 240 frames per second with approximately 4-8 milliseconds of ... | parsec. https://parsec.app/blog/parsec-game-streaming-total-latency-at-240-frames-per-second-c0818cc0daa5, 2022.

[248] Richard S Sutton and Andrew G Barto. *Reinforcement Learning (Second Edition): An Introduction*. MIT press, 2018.

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

[250] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Supporting mobile vr in lte networks: How close are we? *Proc. ACM SIGMETRICS*, 2018.

[251] Jiaxin Tang, Sen Liu, Yang Xu, Zehua Guo, Junjie Zhang, Peixuan Gao, Yang Chen, Xin Wang, and H Jonathan Chao. Abs: Adaptive buffer sizing via augmented programmability with machine learning. In *Proc. IEEE INFOCOM*, pages 2038–2047, 2022.

[252] Mariya Toneva and Leila Wehbe. Interpreting and improving natural-language processing (in machines) with natural language-processing (in the brain). In *Proc. NeurIPS*, 2019.

[253] Martijn van Otterlo and Marco Wiering. *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[254] William N Venables and Brian D Ripley. Tree-based methods. In *Modern Applied Statistics with S*, pages 251–269. Springer, 2002.

[255] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *Proc. ICML*, 2018.

[256] Kurt Wagner. Facebook says video is huge – 100-million-hours-per-day huge. https://www.vox.com/2016/1/27/11589140/, 2016.

[257] Cong Wang, Amr Rizk, and Michael Zink. Squad: A spectrum-based quality adaptation for dynamic adaptive streaming over http. In *Proc. ACM MMSys*, pages 1–12, 2016.

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

[259] Raphael Wimmer, Andreas Schmid, and Florian Bockes. On the latency of usb-connected input devices. In *Proc. ACM CHI*, pages 1–12, 2019.

[260] Keith Winstein and Hari Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *Proc. USENIX ATC*, 2012.

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

[262] Mike Wu, Michael C Hughes, Sonali Parbhoo, Maurizio Zazzi, Volker Roth, and Finale Doshi-Velez. Beyond sparsity: Tree regularization of deep models for interpretability. In *Proc. AAAI*, 2018.

[263] Yikai Xiao, Qixia Zhang, Fangming Liu, Jia Wang, Miao Zhao, Zhongxing Zhang, and Jiaxing Zhang. Nfvdeep: Adaptive online service function chain deployment with deep reinforcement learning. In *Proc. IEEE/ACM IWQoS*, 2019.

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

[265] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *Proc. USENIX NSDI*, 2020.

[266] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *Proc. USENIX OSDI*, 2018.

[267] Gang Yi, Dan Yang, Abdelhak Bentaleb, Weihua Li, Yi Li, Kai Zheng, Jiangchuan Liu, Wei Tsang Ooi, and Yong Cui. The acm multimedia 2019 live video streaming grand challenge. In *Proc. ACM Multimedia*, pages 2622–2626, 2019.

[268] Chuanlong Yin, Yuefei Zhu, Jinlong Fei, and Xinzheng He. A deep learning approach for intrusion detection using recurrent neural networks. *IEEE Access*, pages 21954–21961, 2017.

[269] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proc. ACM SIGCOMM*, 2015.

[270] Haonan Yu, Sergey Edunov, Yuandong Tian, and Ari S Morcos. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp. In *Proc. ICLR*, 2020.

[271] Saman Zadtootaghaj, Steven Schmidt, and Sebastian Möller. Modeling gaming qoe: Towards the impact of frame rate and bit rate on cloud gaming. In *Proc. IEEE International Conference on Quality of Multimedia Experience (QoMEX)*, 2018.

[272] Saman Zadtootaghaj, Steven Schmidt, Saeed Shafiee Sabet, Sebastian Möller, and Carsten Griwodz. Quality estimation models for gaming video streaming services using perceptual video quality dimensions. In *Proc. ACM Multimedia Systems Conference (MMSys)*, 2020.

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

[274] Mo Zanaty, Varun Singh, Ali C. Begen, and Giridhar Mandyam. RTP Payload Format for Flexible Forward Error Correction (FEC). IETF RFC 8627, 2019.

[275] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Proc. ECCV*, 2014.

[276] Jia Zhang, Enhuan Dong, Zili Meng, Yuan Yang, Mingwei Xu, Sijie Yang, Miao Zhang, and Yang Yue. Wisetrans: Adaptive transport protocol selection for mobile web service. In *Proceedings of the Web Conference*, 2021.

[277] Lei Zhang, Yong Cui, Junchen Pan, and Yong Jiang. Deadline-aware transmission control for real-time video streaming. In *Proc. IEEE ICNP*, 2021.

[278] Menghao Zhang, Jiasong Bai, Guanyu Li, Zili Meng, Hongda Li, Hongxin Hu, and Mingwei Xu. When nfv meets ann: Rethinking elastic scaling for ann-based nfs. In *Proc. IEEE ICNP*, 2019.

[279] Songyang Zhang. Soonyangzhang/webrtc-gcc-ns3: test google congestion control on ns3. https://github.com/SoonyangZhang/webrtc-gcc-ns3, 2020.

[280] Wenxiao Zhang, Feng Qian, Bo Han, and Pan Hui. Deepvista: 16k panoramic cinema on your mobile device. In *Proceedings of the Web Conference*, pages 2232–2244, 2021.

[281] Xinyang Zhang, Ningfei Wang, Shouling Ji, Hua Shen, and Ting Wang. Interpretable deep learning under fire. In *Proc. USENIX Security*, 2020.

[282] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. Improving cloud gaming experience through mobile edge computing. *IEEE Wireless Communications*, 2019.

[283] Ying Zheng, Ziyu Liu, Xinyu You, Yuedong Xu, and Junchen Jiang. Demystifying deep learning in networking. In *Proc. ACM APNet*, 2018.

[284] Chao Zhou, Mengbai Xiao, and Yao Liu. Clustile: Toward minimizing bandwidth in 360-degree video streaming. In *Proc. IEEE INFOCOM*, 2018.

[285] Dajiang Zhou, Jinjia Zhou, Xun He, Jiayi Zhu, Ji Kong, Peilin Liu, and Satoshi Goto. A 530 mpixels/s 4096x2160@ 60fps h. 264/avc high profile video decoder chip. *IEEE Journal of Solid-State Circuits*, 46(4):777–788, 2011.

[286] Dajiang Zhou, Jinjia Zhou, Jiayi Zhu, Peilin Liu, and Satoshi Goto. A 2gpixel/s h. 264/avc hp/mvc video decoder chip for super hi-vision and 3dtv/ftv applications. In *Proc. IEEE International Solid-State Circuits Conference*, pages 224–226, 2012.

[287] Dajiang Zhou, Shihao Wang, Heming Sun, Jianbin Zhou, Jiayi Zhu, Yijin Zhao, Jinjia Zhou, Shuping Zhang, Shinji Kimura, Takeshi Yoshimura, et al. An 8k h. 265/hevc video decoder chip with a new system pipeline design. *IEEE Journal of Solid-State Circuits*, 52(1):113–126, 2017.

[288] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proc. ACM PLDI*, 2019.

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

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