实时多媒体传输延迟优化

(申请清华大学工学博士学位论文)

培 养 单 位： 网络科学与网络空间研究院

学　　科： 网络空间安全

研　究　生： 孟 子 立

指 导 教 师： 徐 明 伟　教　授

二〇二三年五月

Latency Optimization in Real-Time Multimedia Streaming

Dissertation Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Doctor of Philosophy

in

Cyberspace Security

by

MENG Zili

Dissertation Supervisor: Professor XU Mingwei

May, 2023

学位论文公开评阅人和答辩委员会名单

公开评阅人名单

刘莹	研究员	清华大学
王文东	教授	北京邮电大学

答辩委员会名单

主席	徐恪	教授	清华大学
委员	谢高岗	研究员	中国科学院计算机网络信息中心
	罗洪斌	教授	北京航空航天大学
	杨家海	教授	清华大学
	刘莹	研究员	清华大学
	徐明伟	教授	清华大学
秘书	王旸旸	助理研究员	清华大学

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容；（3）根据《中华人民共和国学位条例暂行实施办法》及上级教育主管部门具体要求，向国家图书馆报送相应的学位论文。

本人保证遵守上述规定。

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

实时多媒体传输是互联网最重要的应用之一。实时多媒体传输系统对于传输延迟提出了很高的需求。现有的解决方案，在互联网体系结构的各个层次上，均无法很好地满足应用对延迟的需求。其中，延迟波动是延迟优化中最具有挑战性的问题。本文面向实时多媒体传输的延迟波动问题，系统地从互联网体系结构的各个相关层次优化延迟波动。本文主要研究内容和贡献点如下：

1. 针对实时多媒体传输延迟波动组成复杂的问题，本文提出了实时多媒体传输体系结构。该体系结构定位了控制通路延迟和数据通路延迟两部分主要的延迟波动的原因。本文进一步分析了控制通路和数据通路是如何影响最终实时多媒体用户所感知到的延迟的波动，并以此结构为基础，分别针对各个相关组成部分进行分析优化。

2. 针对实时多媒体传输的控制通路延迟波动问题，本文将控制通路划分为反馈和决策两部分。针对这两部分，本文分别提出了 Zhuge 和 Metis，以分别控制这两部分的性能波动。其中，Zhuge 针对控制通路中反馈延迟膨胀，导致发送端不能及时调整发送速率带来的延迟波动的问题，提出了控制通路和数据通路分离的反馈机制，从而提高性能。Metis 针对控制通路中决策算法复杂，导致算法决策延迟或错误带来性能波动的问题，提出了将复杂算法转化为低延迟可解释的决策树，进而减少性能波动。实验表明，Zhuge 和 Metis 能够有效控制由于控制通路不稳定带来的数据通路延迟抖动，并可以在真实路由器上将延迟抖动比例减少最多 75%。

3. 针对实时多媒体传输的数据通路延迟波动问题，本文在数据通路按照应用层、传输层和网络层三个层次分别提出了 AFR、Hairpin 和 Confucius，以控制这三部分的延迟波动。其中，AFR 针对实时多媒体高画质带来的应用处理网络数据不及时的问题，提出对应用层解码器前队列进行主动帧率管理的机制，以减少应用层延迟波动。Hairpin 针对实时多媒体对延迟抖动的极致要求导致现有丢包恢复方案无法满足需求的问题，提出综合重传与冗余恢复的联合丢包恢复方案，以控制传输层因为丢包带来的延迟抖动。Confucius 针对网络层路由器上由于未知的竞争流及干扰带来的延迟波动，提出渐进式主动队列管理机制，来在确保公平性的基础上让路由器在网络层提供稳定延迟。实验表明，AFR、Hairpin 和 Confucius 分别能在不同场景下，将实时多媒体传输延迟抖动降低 13-67%。

关键词：实时多媒体传输；延迟波动；实时性；网络体系结构

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.

Keywords: real-time multimedia transport; latency fluctuation; real-time; network architecture

目 录

插图和附表清单

符号和缩略语说明

ABR	自适应码率，Adaptive Bit-Rate
ABRF	可用带宽减少因数，Available Bandwidth Reduction Factor
ACK	数据确认，Acknowledgement
AP	无线接入点，Access Point
AQM	主动队列管理，Active Queue Management
CCA	拥塞控制算法，Congestion Control Algorithm
DMR	截止时间错失率，Deadline Miss Ratio
ECN	显式拥塞通知，Explicit Congestion Notification
EWMA	指数加权移动平均，Exponential Weighted Moving Average
EWMV	指数加权移动方差，Exponential Weighted Moving Variance
FCT	流完成时间，Flow Completion Time
FIFO	先进先出，First-In-First-Out
FQ	公平队列，Fair Queue
JFI	Jain’s 公平指数，Jain’s Fairness Index
LTE	长期演进，Long Term Evolution
MAC	媒体访问控制，Media Access Control
MCS	调制和编码方案，Modulation and Coding Scheme
MDP	马尔可夫决策过程，Markov Decision Process
MTU	最大传输单元，Maximum Transmission Unit
PLT	页面加载时间，Page Loading Time
PSNR	峰值信噪比，Peak Signal-to-Noise Ratio
QoE	体验质量，Quality of Experience
RED	随机早期丢弃，Random Early Detection
RTC	实时通信，Real-Time Communication
RTT	往返时延，Round-Trip Time
SSIM	结构相似性，Structural Similarity
TCP	传输控制协议，Transmission Control Protocol
UDP	用户数据报协议，User Datagram Protocol
VR	虚拟现实，Virtual Reality
WAN	广域网，Wide Area Network

第 1 章　引言

1.1　研究背景与意义

1.1.1　实时多媒体传输

互联网已经成为了我们生活中不可或缺的一部分。无论我们工作、学习还是社交、娱乐，日常的生产生活活动都要依赖互联网来运行。尤其是近二三十年来，随着网络技术的不断升级，蜂窝网从 2G 逐渐部署到 5G，无线局域网从 WiFi 逐渐部署到 WiFi6，互联网的速度和带宽都得到了极大的提升。这使得互联网的应用场景越来越广泛，从传统的文本、图片等等传输应用扩展到了包含多媒体的传输应用。现在，从城市到农村，人们应该很难想象没有互联网的生活。据统计，2022 年世界上 59.7% 的人口已经在长期使用互联网，每月平均流量使用达到了 49.8GB，平均网速也提升到了 75.4Mbps[1]。

在互联网中，多媒体传输应用是互联网中重要的组成部分。多媒体传输应用包括音频、视频、图像、文本等等多种多样的多媒体数据。早在 2016 年时，多媒体流量便占了互联网总流量的一半。据统计，2022 年，多媒体流量已经占互联网总流量的 82%[1]。特别是自新冠疫情爆发以来，多媒体传输的实时性越来越受到关注。国内的腾讯会议、国外的 Zoom 等软件广泛应用在教学、会议、远程办公等诸多场景。此后，新兴的实时多媒体传输应用也吸引了广泛的关注。实时多媒体传输已经扩展到了云游戏、虚拟现实、远程医疗等等各方面，其面向的对象也从传统的人与人的通话扩展到了人机交互控制等等。这其中，一些常见的场景包括：全息视频会议、云游戏、虚拟现实、远程医疗、工业控制等方方面面。

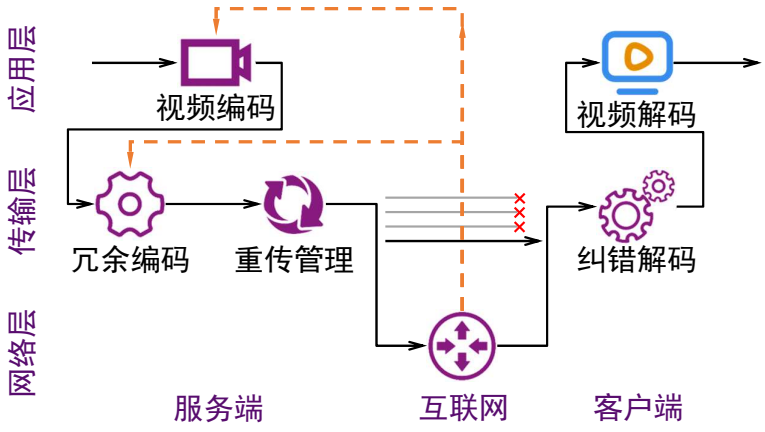


图 1.1　实时多媒体传输的整体结构

总结起来，图1.1展示了实时多媒体传输的整体结构。由于多媒体传输本质上还是网络应用，我们从互联网体系结构的角度出发，在纵向上按照应用层、传输层和网络层对其进行划分，在横向上以服务器、路由器、客户端进行划分。

实时多媒体传输在学术界近年来受到了广泛的关注。网络方向的高水平国际会议诸如 SIGCOMM、NSDI，多媒体方向的高水平国际会议 MM、MMSys 每年均有大量的论文在这个方向上进行优化。工业界，大量的开源闭源框架应运而生。代表性的有谷歌公司维护的 WebRTC，阿里巴巴维护的 AliRTC，声网提供的 AgoraRTC、腾讯的 TRTC 等等。

1.1.2　实时多媒体传输的性能：延迟波动

延迟是实时多媒体传输最重要的指标：它直接与用户体验相关。实时多媒体传输应用不仅需要低延迟，而且还要求延迟的稳定性。例如，假设大多数时间，无线用户可以体验到满意的小于 100ms 的往返时延。但是，如果网络往返时延的第 99 百分位数大于 400ms，那么网络延迟将远远超过应用的延迟预算[2-3]。在这种情况下，每 100 个数据包里面可能就有一个可能会出现高延迟，严重影响用户体验。因此，降低尾延迟，稳定延迟波动对于实时多媒体传输应用来说至关重要。

1.1.2.1　严格的截止时间要求

由于交互式流媒体应用程序持续与人交互，因此控制端到端延迟对于获得较好的用户体验至关重要。例如，视频会议希望端到端延迟小于 130ms[4-5]，而云游戏则希望延迟小于 96ms[6]①。实际上，服务器端和客户端处理通常需要 ≈30 ms[8-11]。因此，**网络**的端到端往返时延不应超过 50-150ms（视应用而定），这便是应用要求的截止时间[12-13]。

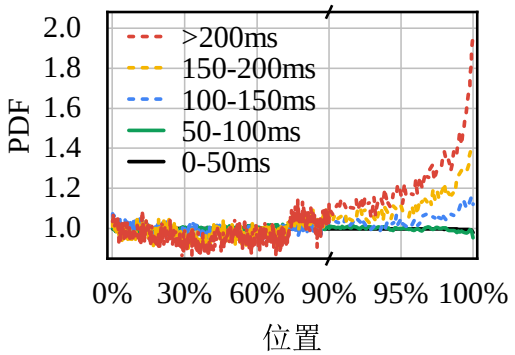


图 1.2　不同延迟的帧的所处位置的分布

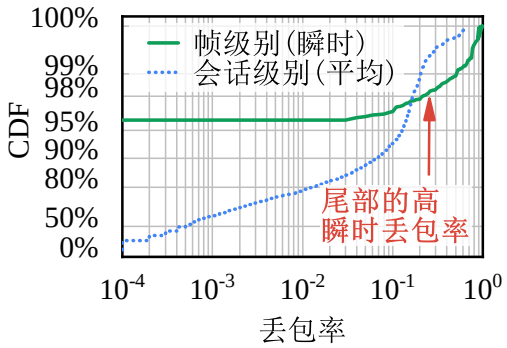


图 1.3　会话级别和帧级别的丢包率分布

本文对一个典型的云游戏服务（腾讯 START 云游戏）也进行了测量。测量

① 这是基于大多数用户的统计数据。不同的用户和应用程序可能对延迟敏感度不同。例如，对于游戏应用程序，3D 游戏对延迟要求比 2D 游戏更严格[7]。

时，每个视频帧的往返交互延迟被归类到几个区间里面。本文由此可以研究用户对不同延迟的容忍度：当用户体验到较高的交互延迟并因无法再容忍这么高的延迟而终止会话时，那些具有高交互延迟的帧将会非常靠近这一用户的会话的结束位置。因此，本测量通过分析具有不同延迟的帧的分布来测量用户对延迟的反应。图1.2中展示了每个类别中那些帧在流中的位置分布，其中 x 轴是该帧在一个会话中的位置，被该会话的全部长度归一化之后的值。例如，位置为 99% 表示该帧出现在会话的非常接近末尾的位置。如果一条线在 0% 到 100% 之间都是水平的（例如，图中的那些实线），这表示这些帧在会话的所有时间都均匀出现。反之，图中的三条虚线表示这些帧更可能出现在会话的末尾。与低延迟帧（实线）的均匀分布相比，延迟大于 100ms（虚线）的帧具有更高的概率出现在流的末尾。由此推论，这是因为用户倾向于在延迟较高时终止会话。这也表明，只要数据包能够在截止时间内（本例中为 ~100ms）传输，那么更快的传输速率就不会对用户体验产生显著影响。

因此，**截止时间错失率**（DMR）应该被尽量降低以实现实时多媒体传输中无缝的用户体验。例如，在该云游戏服务中，交互延迟的截止时间约为 100ms。对于实时多媒体传输，这一截止时间错失率需要被降低到极低的水平才行。例如，即使 DMR 为 10^{-3}，也会导致每 1000 帧就会有 1 帧出现用户体验下降。注意，当帧率为 60fps 时，这一间隔仅仅为短短的为 17 秒。十数秒一次的会严重降低用户的体验[13]。

这与现有工作的关注点是不同的。图1.4给出了一个关于延迟分布（互补累积分布函数，传统工作一般关注于 50 分位（有些也会关注到 90 分位）的较为常见的延迟。然而，10^{-3} 意味着我们需要关注的是 99.9% 分位的延迟，这便提出了全新的要求。

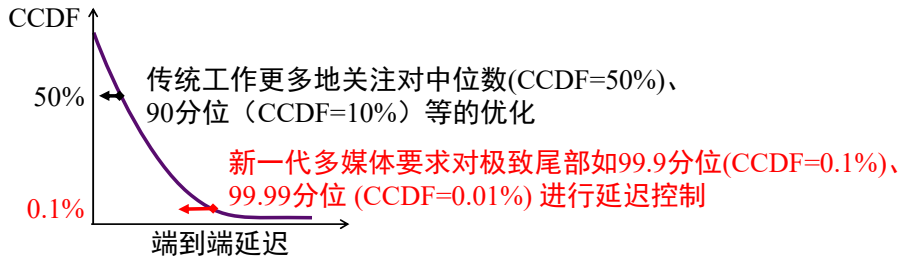


图 1.4　本文更多关注于针对实时多媒体传输的极致尾部的延迟控制优化

1.1.2.2　无法令人满意的性能

然而，当前的网络性能，尤其是无线接入网络性能，在尾部并不令人满意。多个近来的观察结果都支持了这一观点。首先，现有的文献揭示了即使是使用先进

表 1.1　最近一些在无线网络延迟上的相关测量结果

Narayanan 等人 (2020)[14]	5G 跳的尾延迟相比于 4G 并未提升多少，并且可能高达 200ms.
Daldou 等人 (2020)[15]	802.11ax (又称为 WiFi 6) 在有 30 个干扰者的情况下的 WiFi 跳平均延迟大于 30ms.
Bhartia 等人 (2017)[16]	多达四分之一的 802.11ac 的无线接入点会在最后一跳遭受 >100ms 的延迟.
Ghoshal 等人 (2022)[17]	对于中位数用户来说，相比于 4G LTE，5G 毫米波并没有将最大延迟提升多少.

的接入技术，无线网络的尾部延迟也很长。我们在表1.1中总结了近年来的测量结果。即使是 WiFi 6（802.11ax）或 5G（毫米波），无线网络仍然表现不佳。这与内容提供商的一些反馈信息也一致。例如，一家大型电信服务云提供商的技术负责人说，“我们建议客户使用有线网络访问云桌面”。一家云游戏提供商的指南中写道，“当遇到网络问题时，如果可能，请将计算机插入有线以太网连接”[18]。延迟敏感的应用程序发现，由于无线网络的高尾延迟，这些应用的用户会更喜欢不方便但稳定的有线网络。

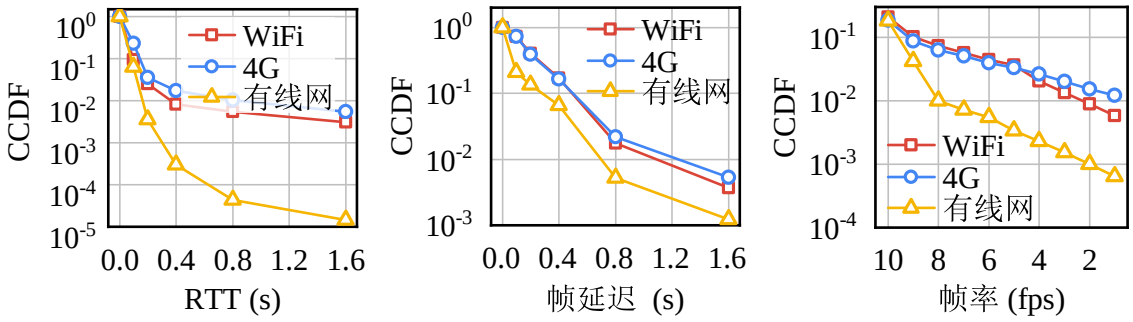


图 1.5　在 WiFi、4G 和有线网上的往返时延、帧延迟和帧率分布

此外，本文的测量结果也揭示了无线网络的尾部性能下降。本文测量了一个每天服务数百万用户的在线实时通信服务，并展示了有线网、WiFi 和 4G 接入网络的网络条件和应用性能。数据来源于**一个每天服务数百万用户的在线实时通信服务**。帧延迟是指应用层测量的延迟。如图1.5所示，大多数时间，无线网络可以提供满意的往返时延（<100ms）。但是，无线网络的第 99 百分位往返时延大于 400ms，远远超过应用的延迟预算[2-3]。在这种情况下，100 个数据包中的一个可能会出现高延迟，严重影响用户体验。应用层指标也表现出类似的模式：无线用户遇到的视频延迟（长帧延迟）是以太网用户的 2 倍。此外，无线网络的帧率下降（视频停顿）比率是有线网络的 10 倍。

1.2　研究内容

本文以互联网体系结构为起点，对新一代多媒体传输的延迟波动来源进行了全面分析。本文完成了从发现问题、定义问题到解决问题的完整过程：首先，本文以新一代多媒体传输的延迟波动为研究对象，对其进行了全面分析，发现了互联网体系结构中的多个环节对延迟波动的影响。其次，本文针对这些环节中延迟波动产生的原因分别进行优化，从协议栈的应用层到网络层全面地进行了深入研究。

下面对本文中各个部分的主要内容进行简要介绍。

1.2.1　相关研究综述

实时多媒体传输应用是互联网中重要的组成部分，其在网络中也已经被研究了数十年。从学界到业界、从国内到国外的解决方案层出不穷。自疫情以来，实时多媒体传输应用的使用量大幅增长，也涌现出了许多最新的研究工作。然而，针对这些较新的研究工作，目前还没有非常系统的最新研究进展综述。本文在总结现有的研究工作时，按照互联网体系结构的各个层次对这些新的工作进行了整理和分析。本文从控制通路、数据通路对现有工作进行了分类，并指出了一些现有研究的缺陷。具体工作在第2章中展开。

1.2.2　实时多媒体传输架构

在分析实时多媒体传输的延迟问题之前，首先需要了解延迟的产生来源。本文系统地提出了实时多媒体传输中延迟波动可能的产生来源，其中提出了**控制通路**和**数据通路**两部分延迟对整体端到端延迟的影响。本文首先对一个视频帧从渲染到播放的生命周期进行了逐组成部分的分析，并进行了形式化的表达。这里，还梳理了不同组成部分之间是可能如何相互影响的。通过对实时多媒体传输架构的整体分析和建模，本文后面才可以对逐个组成部分进行优化。

这其中，控制通路指的是控制信息传递的通路，尤其是指当网络中出现性能波动，到相应的端上做出响应的动作这一控制回环。如果端上对于网络波动响应得太晚，也会导致由于带宽的不匹配而引起的延迟波动。数据通路则是指一个数据包本身会经过的通路，包含应用层、传输层、网络层的各个组成部分。如果某个环节出现了延迟波动，最终总的数据包端到端延迟也会随之波动。具体工作在第3章中展开。

1.2.3　控制通路延迟

在控制通路延迟的分析与优化中，本文主要关注反馈和决策这两个组成部分。这两个组成部分的延迟波动以及可靠性都会直接影响数据包的端到端延迟。

反馈指的是网络中的“波动发生”这一信号到达发送端的过程。当某一时刻网络可用带宽发生了变化，发送端没办法像上帝一样在瞬间了解到这种变化 — 这一消息的传递需要时间。例如，在 TCP 协议中，消息通常通过 ACK 包的延迟变化或送达速率变化等来进行传递。这时，反馈的时间便至少需要一个往返时延。具体工作在第4章中展开。

决策指的是从发送端首次接收到网络波动发生后的信号，到发送端作出响应的过程。当发送端只观测到了一两个网络波动后的信号，它可能不见得会当真 — 网络的噪声很大，一两个包的变化可能会让发送端以为是噪声。例如，很多拥塞控制算法都会在连续一个 RTT 甚至更长时间都观测到某种网络变化后，才会有信心做出决策。这时，决策的及时程度以及决策的可靠性便十分关键了。具体工作在第5章中展开。

1.2.4　数据通路延迟

在数据通路延迟的分析与优化中，本文主要关注应用层、传输层和网络层三个在网络体系结构中偏上层的组成部分。这几层中每一层的延迟波动也都会直接影响数据包的端到端延迟。

应用层延迟一般包括应用对数据的处理以及在应用层的排队时延。例如，在接收端，当数据已经被传输层排好序到达接收端后，如果上层应用并不能够及时处理并取走这些数据，这些数据便不得不在应用层排队等待处理。在这种情况下，如果本身应用对端到端的延迟要求极严格（例如，实时多媒体传输），那这些排队便会导致端到端延迟的升高。这一问题在随着实时多媒体传输要求的清晰度和帧率增高而变得日趋严重。应用层的延迟便因之需要进行主动管理。具体工作在第6章中展开。

传输层延迟则主要是由于丢包恢复带来的。一个数据包在传输的过程中可能会丢失：无论是队列溢出还是无线网络干扰，都有可能导致该数据包损坏而无法正常通过校验到达接收端。在此时，发送端便需要及时对这一丢包事件进行识别，并对丢失的数据包进行重传。然而，无论是对丢包的识别（例如，乱序和丢包可能表现很像），还是对丢失的数据包进行重传的过程均会产生额外的延迟。同时，由于互联网设计的“尽力而为”的本质，这种丢包又通常是难以避免的。这一问题在随着实时多媒体传输的延迟要求变得日渐严格而加剧。因此，降低传输层延迟同

样是必要的。具体工作在第7章中展开。

网络层延迟则一般由队列的速率和带宽不匹配造成。在网络的队列中，多个用户的不同流量会共享（或竞争）同一资源 — 带宽。而其他用户的流量特性又往往是当前用户难以提前预料的。例如，如果与某个实时多媒体传输的流竞争的其他用户突然增大了发送速率，那么该实时多媒体传输流量的可用带宽便一般也会因之减少。当然，拥塞控制等算法会随即收敛到新的稳态中去。但随着对于尾部的关注越来越高，哪怕是这一收敛过程的波动都会导致端到端延迟对应用户体验的下降。具体工作在第8章中展开。

1.3　主要贡献

结合着上一节中对总体研究内容的概述，本文的主要贡献包含如下五个部分。

1. 在控制通路的反馈环节，提出了缩短反馈回环的拥塞信号的解决方案。本章详细分析了当前网络中，端到端的拥塞控制、速率控制等机制是如何对网络波动作出响应的。本章通过大量的实验验证了当反馈回环膨胀时，数据通路上的延迟波动会随着反馈回环的膨胀而增大。这在广域网远距离传输（例如视频会议）中尤为明显。针对这一现象，本章提出了一种缩短反馈回环的拥塞信号早反馈的解决方案，通过将反馈回环与数据通路进行解耦，从而实现了缩短反馈回环的拥塞信号早反馈的目的。具体而言，该工作通过对实时多媒体传输协议，根据其反馈模式分类为带内反馈与带外反馈，并针对性地对不同类型的反馈模式分别进行了优化。基于真实路由器和大规模仿真的实验表明，本章提出的缩短反馈回环的拥塞信号早反馈的解决方案可以有效地减少端到端延迟波动，从而提高了用户体验。

2. 在控制通路的决策环节上，提出了决策延迟低、结果稳定的速率控制决策框架。本章侧重于分析，随着深度学习、整数规划等复杂的速率控制决策算法的引入，端到端的速率控制决策算法的复杂度也在不断增加，从而导致了端到端的速率控制决策算法的计算开销也在不断增加。本章通过实验说明了，当决策黑盒化且决策耗时不断增加时，随着可能的决策滞后与错误，这些复杂的算法可能会在端到端的速率控制决策中引入更多的性能波动。针对这一现象，本章提出了一种轻量、可靠的速率控制决策的转化与解释框架，通过将优化后的复杂的速率控制决策算法转化为简单的速率控制决策算法，从而实现了决策的及时性与可靠性。具体而言，该工作将现有基于机器学习、整数规划等复杂的速率控制决策算法转化为基于决策树简单的速率控制决策算法。基于现有算法的实验与分析表明，本章提出的轻量、可靠的速率控制决策的转化与解释框架可以有效地减少性能波动，从而提高了用户体验。

3. 在数据通路的应用层上，提出了通过自适应帧率调整来降低应用层排队延迟的解决方案。本章主要分析了，随着新一代多媒体应用的出现（例如云游戏），应用对多媒体传输的画质要求越来越高，从而导致了应用层中视频编解码器的延迟也在不断增加。本章通过结合真实应用大规模测量表明，当应用层中视频编解码器的延迟波动较大时，数据通路上的延迟波动也会随着应用层中视频编解码器的延迟波动而增大。由于现有的应用层设计缺乏主动队列管理，上述延迟还容易被进一步放大。针对这一现象，本章提出了一种自适应帧率调整的解决方案，通过主动调整应用层中视频编解码器的帧率，从而实现了应用层中视频编解码器的延迟波动的减小。具体而言，该工作基于网络状况和应用状况的联合分析，通过排队论及随机过程的建模分析，提出了一种基于应用层的主动队列管理的解决方案。面向大规模用户的实验表明，本章提出的自适应帧率调整的解决方案在云游戏应用中可以有效地减少端到端延迟波动。

4. 在数据通路的传输层上，提出了针对实时多媒体传输的丢包恢复机制。本章着重分析了，随着新一代多媒体应用对延迟波动的容忍度越来越低，现有传输层的丢包恢复机制已经无法满足应用对延迟波动的需求。基于真实测量数据的分析表明，在现有传输层的丢包恢复机制以及当前网络环境下，仅仅依靠重传或是冗余的单一丢包恢复机制几乎无法达到要求。针对这一现象，本章提出了一种综合现有丢包恢复机制，特别是重传和冗余恢复这两种机制，的联合丢包恢复方案。具体而言，该工作采用马尔科夫链对丢包及重传进行联合建模，提出了一种最优的添加冗余以及决定是否重传的策略。基于真实网络数据集的实验表明，本章提出的联合丢包恢复方案可以有效地减少端到端延迟波动，同时也降低了不少带宽开销的成本。

5. 在数据通路的网络层上，提出了能够抑制多应用竞争排队、稳定性能的路由器队列管理方案。本章指出，尽管在端侧有诸多方案试图控制延迟波动，但是在数据通路上，由于网络层多应用突发竞争排队带来的端到端延迟波动仍然是一个严重的问题。无论端侧如何优化，对于网络内其他用户的突发竞争端侧没办法做到未卜先知，因此依然需要对网内瓶颈路由器进行管理上的优化，从而减少端到端延迟波动。基于千余个网站的测量表明，由网页应用带来的端到端延迟波动是一个严重的问题，尤其是针对新一代多媒体应用这种对延迟波动有较高要求的应用。针对这一现象，本章提出了一种新的路由器队列管理方案，通过对带宽分配在不依赖于端侧信息的情况下进行差分服务优化，从而减小端到端延迟波动。具体而言，该工作通过观察不同流对瓶颈队列的侵占情况，来推测出不同流的延迟敏感性，从而实现了对不同流的差分服务优化。基于真实路由器及上千个网站的

测试表明，本章提出的路由器队列管理方案，在不依赖任何端侧标签等信息的情况下，可以有效地减少端到端延迟波动。

1.4　论文组织结构

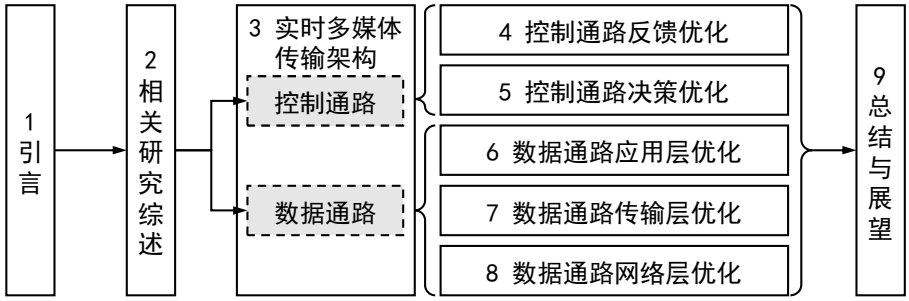


图 1.6　本文组织结构

本文由九个章节组成，其整体结构如图1.6所示。

第1章为引论，介绍了本文的研究背景、研究内容、主要贡献以及论文组织结构。

第2章为相关工作，介绍了本文中涉及的相关工作。

第3章为本文梳理分析的现有实时多媒体传输的整体架构，并分析了哪些组成部分的延迟会对最终的用户感受到的延迟产生影响。在此，这一章指出，控制通路中的反馈和决策延迟，以及数据通路中的应用层、传输层、网络层延迟，都会对最终用户感受到的端到端延迟产生影响。

第4到8章对上一节中提到的两个通路五个问题分别进行了研究。

在控制通路方面，第4和5章分别研究了控制通路中两个组成部分带来的延迟波动：第4章介绍了针对控制通路反馈延迟膨胀带来的端到端延迟波动，如何通过减少反馈回路来减少延迟波动的方案；第5章介绍了针对控制通路决策不稳定及耗时长带来的端到端性能波动，如何通过简化模型来减少性能波动的方案。

在数据通路方面，第6到8章分别研究了数据通路中三个组成部分带来的延迟波动：第6章介绍了针对应用层编解码器性能波动带来的端到端延迟波动，如何通过自适应调整帧率来减少延迟波动的方案；第7章介绍了针对传输层丢包及其恢复机制带来的端到端延迟波动，如何通过综合多种丢包恢复机制的联合恢复方案来减少延迟波动的方案；第8章介绍了针对网络层多应用突发竞争排队带来的端到端延迟波动，如何通过对带宽分配在不依赖于端侧信息的情况下进行差分服务优化来减少延迟波动的方案。

最后，第9章为结论，总结了本文的研究成果，并展望了本文未完成的一些研究方向。

第 2 章　相关研究综述

本章概述了现有的多媒体传输系统的相关的研究，尤其着重于从网络方面出发，针对多媒体传输进行优化的相关工作。本章遵循前面引言中，按照互联网体系结构对现有技术进行的划分，来分别讨论各个层中现有工作是如何改善实时多媒体传输的性能，或广义上改善网络应用的延迟的。

如图2.1所示，本章将主要介绍虚线框内的应用层、传输层、网络层的工作。这是网络研究者主要关注的部分。在链路层和物理层的工作更多是通信领域研究者的研究内容。在这两层中针对低延迟的工作以及针对实时多媒体的工作在本文中不予展开。本章在接下来的几节中，对比现有工作，阐述了本文的创新点。

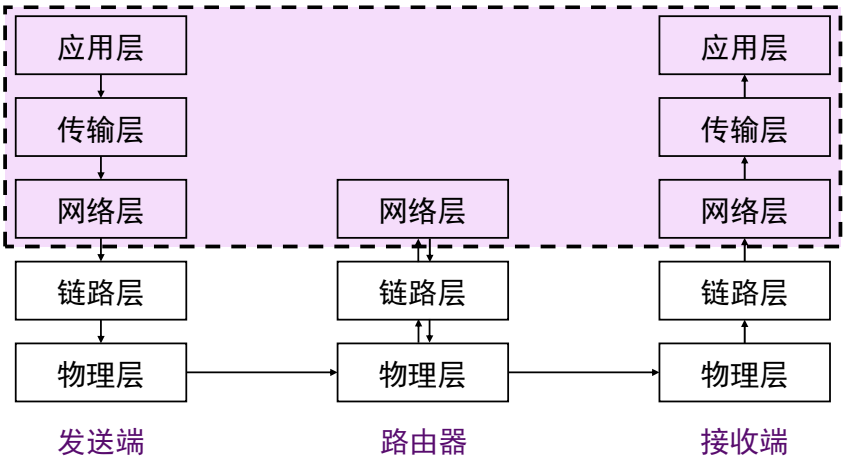


图 2.1　互联网体系结构及本章主要关注的相关工作

第2.1节介绍了目前针对多媒体传输在应用层的各种优化工作。第一类为针对编解码器的优化工作，包括提出新的编解码算法等，使其在波动的网络链路上取得稳定的性能。第二类为针对多媒体传输特性所进行的协议设计的工作，以更好地适配需要传输的多媒体内容。第三类为结合网络状况对应用进行自适应调整的工作，例如自适应码率调整等，以提升用户体验。

第2.2节介绍了现有互联网上在传输层试图优化延迟及延迟抖动的工作。传输层的两个功能即为速率控制及可靠传输。第一类为以拥塞控制为代表的近年来在速率控制上对互联网延迟优化的工作。第二类为以丢包恢复为代表的在可靠传输上针对多媒体传输的延迟优化工作。

第2.3节介绍了在网络内的网络层控制延迟的工作。网络层的主要设备即是网络路由器。第一类工作便是介绍如何调整路由器上的缓冲区大小来控制延迟。第二类工作介绍如何通过对路由器上的队列进行主动管理来和端算法配合以控制延

迟。第三类工作则直接介绍了如何通过端网间的直接消息传递来进行性能优化。

2.1　数据通路应用层相关工作

表 2.1　应用层中的实时多媒体优化相关工作

学界/业界方案	解决方案	主要思路
Swift (NSDI'22)[19] CGEncoder (MMSys'20)[20] VP9 (Google'13)[21]	实时编解码器	通过更新编解码设计在弱网等情况下依然保证内容解码
BB (SIGCOMM'14)[22] Pensieve (SIGCOMM'17)[23] Puffer (NSDI'20)[24]	自适应码率算法	通过让码率适配带宽，以降低卡顿
RTP/RTCP (RFC8888)[25] RTSP (RFC7826)[26] DTP (ICNP'21)[27]	多媒体传输协议	通过协议设计来传递应用需要的信息

实时多媒体传输应用主要包含以下几个主要功能。一是当视频源产生了画面之后，编码器需要将画面编码为视频流。例如，视频会议的画面从用户端的相机中获取，云游戏及虚拟现实等的画面由 GPU 渲染而成。这都需要将原始视频进行编码才能够继续传输。二是在视频编码的过程中，编码的参数（主要是码率）需要根据网络状态来进行实时调整。例如，当网络状态变好时，编码器可以适当提高编码码率，以为用户送达更加清晰的视频内容。而类似地，当网络状态变差时，编码器也会降低码率，以确保起码用户能够看到不卡顿的内容。三是当内容准备好后，发送端还需要将这些内容通过应用特有的协议进行发送，以便应用更好地对内容进行管理。如本章引言所述，针对多媒体传输，在应用层及与应用联合设计方面，近年来的工作越来越多。我们将从下面几个方面，分别进行介绍。

2.1.1　编解码器优化

编解码器的发展历史很长，应用范围也很广。视频编码的主要原理是利用视频内容的时间相关性和空间相关性，大幅通过存储差分值等方式压缩内容以节约带宽成本。目前比较广泛使用的编码器以 H.264 为主[28]。近年来，编解码器的优化工作主要集中在两个方面，一是优化编码器的性能，二是针对应用的具体场景对编码器进行针对性地设计和优化。在编码器本身的性能优化上，近年来提出了新一代的 H.265 编解码机制[29]，其能够在同样清晰度下节省大量的带宽成本。然而，由于这一新的机制涉及专利权等诸多问题，目前部署的情况远不如 H.264 理想。同时，最近还有研究者在推动 H.266 编解码的标准化与示范性应用。而在实时多媒体传输领域，目前使用较为广泛的是由谷歌公司推动部署的 VP9 编码器[21]。

其现在已经被包含在 WebRTC 实时音视频传输框架内，可以方便地被开发者使用。

此外，还有其他的研究工作，专注于优化其他应用指标，如图像或视频质量（例如 SSIM[30]或 PSNR[31]）。例如，Alfalfa[32]针对实时多媒体传输的转码过程中大量用户的多线程并行进行了特殊优化。Salsify[33]则进一步让编码器注意到网络状态，并为网络预留了可以调整的空间。最近的工作 Swift[19]更是利用神经网络等技术进一步优化编解码器的编码效率及处理延迟，让用户能够更加流畅地使用虚拟现实（VR）等技术。CGEncoder[20]则是结合了云游戏等游戏应用的特点，首先分析了游戏用户的具体需求——游戏用户的用户体验可能不见得完全与 PSNR 或 SSIM 等清晰度的客观指标完全一致。其基于此进一步设计了编解码机制使得编解码器更适配于云游戏的场景。上述工作与本文关注的交互延迟（每帧的延迟）是正交的，他们关注的是视频的清晰度，而我们关注的是用户可能会有的交互滞后。因此，上述清晰度等的优化完全可以与本工作的延迟优化共存。

同时，设计编解码器最大的问题是可部署性的问题。视频的解码又对实时性有着很强的需求：当一个视频帧播放完成时，下一帧应该已经完成解码准备播放才不至于使用户感受到卡顿。然而，编解码包含大量的数学运算，其在 CPU 内是极其消耗资源的。因此，现有解决方案中，一般在 CPU 或显卡内均有专门的编解码芯片来进行处理，以加速解码过程。但是，这些硬件解码芯片不见得支持上述提到的新兴编解码机制。事实上，许多工作自身谈及的一大不足便是不能被现有硬件支持。因此，尽管 H.264 编码机制已经被提出了 20 余年，其仍是目前应用最为广泛的编解码机制。

2.1.2　自适应码率优化

正如前面引言所述，自适应码率是现有多媒体传输机制中的几个重要的组成部分之一。其主要功能是针对网络状况波动对编码器的编码码率进行修改，以确保编码器的码率不要超过网络的承载能力。在实时多媒体传输诞生的时候开始，便有了相应的自适应码率算法。最近十余年来，比较典型的算法是由网飞公司提出的 Buffer-based 算法[22]。其创新性地提出，通过估计在点播多媒体视频中客户端缓冲区的情况，来调整网络中传输的多媒体码率。当客户端缓冲区较低时，这意味着客户端卡顿的风险增高，此时就需要降低编码码率以使得新的内容尽快传送到客户端。当客户端缓冲区较高时，这意味着编码器可以尝试探索下更高的编码码率，以为用户提高更好的画质。在这一方向上，还有类似 BOLA[34]等算法，同样基于缓冲区的占用情况进行决策，但同时又能够基于李雅普诺夫稳定性进行理论分析。BOLA 算法目前是应用最广的点播流媒体框架 dash.js 的默认自适应码率算法。在此之后，还有 BOLA-E[35]等算法进行了进一步改进与提升，进一步对基

于缓冲区的方法进行优化。

除此之外，还有发送速率的估计算法，比较有代表性的是 PANDA[36]与 Squad[37]。其类似于拥塞控制，根据网络状态来估计出当前比较适合传输的视频码率。并行地，还有许多混合基于缓冲区与基于网络状态的自适应速率控制算法。例如，有学者提出采用整数规划对自适应码率问题进行系统性建模，并提出 RobustMPC 算法[38]进行优化求解，得到最适合当前传输的码率决策。近年来，更是出现了使用机器学习算法来优化自适应码率算法的热潮。SIGCOMM 2017 会议上提出的 Pensieve[23]是首个采用深度神经网络来优化自适应码率选择的工作。其采用深度强化学习，对自适应码率问题进行了建模，并设计了相应的状态空间、动作空间及奖励函数，采用一系列算法对自适应码率算法进行优化。随后，还出现了 HotDASH[39]等工作对神经网络的结构、优化方法等进行进一步优化，以提升用户的体验。这方面的工作也一直是学术界近年来的热点。然而，目前的算法[23,38]与一些分析给出的理论最优仍然存在较大的差距。因此，我们认为还有很大的改进空间。作为一个例子，学术界不断举办比赛寻求更好的 QoE 算法[40]。总之，为了更好的性能，不断优化的方法已经（并将继续）被提出[40]。

本工作在应用层并不关注于自适应码率工作，即我们之间同样是正交的。这是因为，自适应码率工作是类似于拥塞控制那样，通过调整应用端速率以避免网络或客户端出现拥塞或卡顿。而在本文中，本工作更多地关注的是如何缩短本身就在应用层中的那些延迟。因此，现有的这些算法在本文第1章提出的架构中，但并未针对延迟抖动，或极致尾延迟优化。

2.1.3　多媒体传输协议设计

另一种近年来重要的研究工作，是设计新的多媒体传输协议。应用层协议在互联网体系结构中，也是不可或缺的。在传输层协议之上，需要应用层也有相应的协议才能确保内容的正确传输。最近应用最广的协议是 RTP/RTCP 协议[25]。RTP 和 RTCP 是一对基于 UDP 的协议，其中 RTP 协议从服务器向客户端发送数据包以传输视频内容，所以其是数据通路的协议；RTCP 负责从客户端向服务器反馈网络状态、视频状态等状态信息，所以其是控制通路的协议。RTCP 协议会构造发送端报告（Sender Report，SR），接收端报告（Receiver Report，RR），来报告发送端及接收端信息。RTCP 协议还会构造 NACK（Negative Acknowledgement）、TWCC（Transport-wide Congestion Control）报文以向发送端报告丢包及延迟情况。发送端可以选择性地决定是否要重传某些数据包。在这种情况下，这样一个基于 UDP 的应用层协议其实也基本可以实现几乎可靠的传输。无论是开源框架 WebRTC[41]，还是工业界的解决方案如 Zoom[42]或 Google Meet[43]，均采用了这种协议或其变

种进行传输。

在历史上，还有 RTSP[26]等协议针对多媒体传输应用进行传输。这些协议基于 TCP 等可靠传输协议，以免去了在应用层确保传输可靠性的开销。同时，最近还有一些学者注意到新兴应用中对于延迟的敏感需求，提出了诸如基于截止时间的传输协议（Deadline-aware Transport Protocol，DTP）[27,44]，来在协议设计中，搭载对应数据报文的截止时间信息。在这种情况下，无论是网内设备还是接收端，均可以根据数据包的截止时间信息进行更合理的数据包调度，以使得最终交付的数据包能最大化地满足应用的需求。

相比之下，本文并未提出新的应用层协议，而是在现有框架基础上追求可靠低延迟。这是因为，从可部署性的角度出发，本文希望在数据通路上能够尽量和现有框架适配、共存，以使得提出的解决方案尽量有实际应用的价值。同时，现有的协议设计本身对应用的延迟也并未显式地提出需求。本文在应用层上的工作事实上几乎可以与所有常见的应用层协议共存。

2.2　数据通路传输层相关工作

表 2.2　传输层中的实时多媒体优化相关工作

学界/业界方案	解决方案	主要思路
Sprout (NSDI'13)[45] GCC (MMSys'16)[46] NADA (RFC8698)[47] SCREAM (RFC8298)[48] Copa (NSDI'18)[49] Vivace (NSDI'18)[50]	拥塞控制	通过让发送速率适配带宽 以减少延迟
WebRTC (ICIP'13)[51] AdaptFEC (MM'19)[52] StreamMelt (NSDI'23)[53] TLP (RFC8985)[54]	丢包恢复	通过减少重传来减少端到端延迟

传输层是互联网优化中，尤其是 SIGCOMM/NSDI 所代表的网络社区中，受关注非常多的一个组成部分。传输层主要的功能就是确保应用层交付给其的数据能够及时可靠地到达接收端。这在延迟出现变动、可用带宽波动、以及网络的丢包情况时刻发生变化下尤为难得。因此，传输层的两个主要功能即是速率控制与可靠传输。速率控制功能在实际互联网中以拥塞控制为主，而可靠传输则主要以丢包恢复为主。速率控制更多地在宏观尺度上，尝试通过调整拥塞窗口等手段，避免互联网中出现长时间的排队并确保互联网的效率。而可靠传输则更多地在微观尺度上，针对某一个或某几个丢失的数据包，将它们能够送达到接收端。下面，我

们便将从这两方面对现有工作中目标与实时多媒体传输接近的工作进行简要介绍。

2.2.1　拥塞控制

拥塞控制至今已有四十余年的历史，在此不再赘述。其中，低延迟拥塞控制受网络研究者关注已久。早先的拥塞控制算法如 Reno[55]、Cubic[56]等方法是通过尽可能挤占队列的方式来提升网络资源利用率，但这会导致端到端延迟的上升。近年来应用比较广泛的是谷歌公司 2016 年提出的 BBR[57]算法。BBR 通过测量链路的瓶颈带宽与往返时延来判断到底有多少数据报文应该在网络中，并以此速率为发送速率来发送数据包。这样，BBR 不再需要挤占瓶颈队列，因此可以获得较低的延迟。除此之外，还有 Sprout[45]算法、Verus[58]算法、Copa[49]算法等进一步利用延迟信息等针对 TCP 协议进行更为精准的速率控制。像 Verus[58]是专门针对蜂窝网络的信道波动性而对延迟估计做出适应性调整的拥塞控制；Copa[49]是利用延迟波动的信号对端上的拥塞窗口进行调节。他们均能够较为有效地获得较低的延迟。

在实时音视频领域，上述算法也得到了一定的部署。例如，脸书公司在其直播业务中采用 Copa 算法进行测试[59]，并取得了较好的效果。除此之外，也有许多专门针对实时音视频应用的拥塞控制算法被提出。例如，谷歌公司提出了 GCC[60]算法，应用在 WebRTC 框架中。GCC 算法通过利用延迟梯度（delay gradient）的信息来控制发送速率——测量每个数据包的延迟通常是不准确的，因为区分排队延迟和传输延迟一直是端到端拥塞控制算法的难题。因此，GCC 算法关注两个数据包的延迟的差——其称之为延迟梯度——来进行速率控制：当数据包延迟在逐渐增加时，网内的瓶颈队列大概在堆积的过程中。这时，GCC 便会降低其发送速率，反之亦然。除此之外，思科公司和爱立信公司也分别提出 NADA 算法[47]和 SCREAM 算法[48]来专门针对实时音视频传输进行优化。他们进一步利用了一些包含显示拥塞控制（Explicit Congestion Notification，ECN）等信息，来降低端到端传输的延迟。

然而，如我们在第1章介绍的那样，即使是拥塞控制在管控延迟抖动上做出了许多努力，现有算法依然难以做到令实时多媒体传输应用满意的延迟。用户依然在很多情况下经受着糟糕的网络体验。这一方面当然是因为应用对延迟、卡顿的需求变得越来越高，另一方面也说明了单纯的端到端拥塞控制算法优化的局限性。本文正是试图探究在传输层以及拥塞控制上，在现有工作的基础上，新的性能瓶颈出现在哪里。

2.2.2　丢包恢复

丢包恢复是网络传输中的一个重要问题，它的目的是在网络出现丢包时，通过恢复来保证传输的可靠性。TCP 协议区别于 UDP 协议的一个显著性特点便是其能够在内核中有效地对丢失的数据包进行恢复。针对包括实时多媒体传输在内的绝大部分应用，当网络中出现了丢包时，主要的恢复方式是对这个数据包的重新传输。重传数据包也有许多设计需要进行考量，最主要的是如何判断一个数据包已经丢失，而不是乱序、延误等。TCP 初始的设计是采用 RTO（重传超时）来进行判断——当等了一段时间（一般为 1 秒钟或 200 毫秒）依然没有收到原来数据包的确认包后，发送端会选择对这一数据包进行重新传输[61]。随后，快速重传机制的提出让连续 3 个相同的确认包即可快速触发重传。近来的工作则是提出了 TLP 尾丢弃探测[54]等机制，在等 3 个相同的确认包耗时较长的情况下，依然能够及时地将丢弃的数据包进行重传。

另外一条线上的研究便是通过引入冗余的方式来进行丢包恢复。这种方式也较容易理解：例如，当发送端打算发送 3 个数据包时，由于发送端担心数据包可能丢失，因此发送端可以采用前向纠错码（Forward Error Correction，FEC）的方式编码出来第 4 个数据包，并将这 4 个数据包一并发出。此时，只要接收端接收到任意 3 个数据包，其均有能力将第 4 个数据包恢复出来。在这一方向上，一种做法是采用现有的 FEC 技术，但根据当前网络状态动态调整其参数：当网络丢包率变高时，冗余包的比例就多些。例如，Bolot[62]和 USF[63]算法分别根据历史丢包恢复的情况，视其恢复能力进行参数调整。在这个方向上，后续还有 WebRTC 的 FEC 参数策略[64]甚至近年来采用深度强化学习等机器学习工具进一步预测网络状态并优化冗余参数的算法[65]。他们在不同的实验测试环境中，均能取得较好的性能，有效地对数据包丢失的情况进行恢复。

除了对冗余的参数进行优化外，另一类工作便是直接对冗余编码的机制进行设计。这一般需要较强的群、环、域等数学相关的知识。这其中的许多工作也发表在信息论相关的期刊上，例如 *IEEE Transactions on Information Theory*。比较有代表性的是 AdaptFEC[52]，Fong et al.[66]的编码机制、Krishnan et al.[67]的编码机制等。然而，这些算法由于其复杂度实在较高，在实际部署中困难较多。事实上，目前在实时多媒体传输中应用较为广泛的冗余编码机制是异或码。甚至一些稍复杂的如 RS 码应用都尚不成熟。

本工作在传输层上针对丢包恢复这一场景的主要贡献在于将冗余与重传这两类工作进行联合优化。从实用性的角度来说，本工作未提出新的冗余编码机制，而是尝试利用现有编码机制进行参数优化。本工作另一点突出之处便是从延迟波动

的角度对丢包恢复的机制进行优化。这些内容将在本文第7章中进行详细介绍。

2.3　数据通路网络层相关工作

表 2.3　网络层中的实时多媒体优化相关工作

学界/业界方案	解决方案	主要思路
CoDel (CACM'12)[68] RED[69], BLUE[70], GREEN[71], Yellow[72]	主动队列管理	尽早丢包迫使发送端降速 以避免超发
BDP/n (SIGMETRICS'21)[73] ABS (INFOCOM'22)[74]	队列大小优化	设置合适的队列大小 以降低延迟
XCP (SIGCOMM'02)[75] RCP (INFOCOM'08)[76] Kickass (ICNP'16)[77] ABC (NSDI'20)[78]	端网消息传递	携带更多维度的网内状态 以便决策

广域网中网络层的优化工作在近年来并不是很多。这主要原因是网络层的主要组成部分是网络内部的路由器。在数据中心等其他场景下，路由器（或交换机）的更新换代较为频繁。因此，新的技术有机会能够得到较快的部署。而在广域网下，几乎不存在一个实体控制一条路径上所有设备的情况。因此，在下面讨论的工作中，其可部署性是我们关注的很重要的一点。

而在路由器上，其能够做的事情就是对流经路由器的数据包进行操作，以隐式或显式地告知发送端当前的网络状态。基于此，隐式地告知发送端可以通过主动队列管理的技术来达到低延迟——当网络状态恶化时，路由器可以选择性地丢弃一些包；也可以通过直接调整队列大小来从物理上限制其延迟的最大值——如果缓冲区太小，数据包不得不被丢弃，这样尽管丢包率可能上升，但延迟也能够有界，而这对实时多媒体不见得是坏事。还可以通过显式地构造新的网络层协议，将网络状态捎带回发送端以达到信息传递的目的。本小节将从这几方面对这些工作分别进行综述。

2.3.1　主动队列管理

在网络层，主动队列管理（Active Queue Management，AQM）是一种来控制网络的拥塞的常用方法。路由器上有许多 AQM 算法。较早的主动队列管理算法是 RED[79]，其通过在早期以概率性随机丢弃的方式来告知当前网络恶化的情况。目前在许多边缘路由器上部署的默认主动队列管理算法是 2012 年提出的 CoDel[68]，其主要解决的问题是根据队列长度的估计难以适应不同带宽的路由器，而相比之下采用在队列中的停留时间可以更精准地控制延迟目标。除此之外，还有更多的

AQM 算法被提出，如 SFB[80]、Green[71]、Yellow[72]、Black[81]和 AFD[82]。最新的进展是 2023 年刚刚成为 IETF RFC 的 DualQ 算法[83]，其是 IETF 的 L4S 工作组的一部分，通过将数据流区分成不同的类别然后分别进行主动队列管理。

除此之外，在数据中心中，同样有大量的工作对数据中心交换机的队列进行管理。例子包括 PIAS[84]、pFabric[85]以及 SIGCOMM 2022 的 ABM[86]等算法。然而，这些工作与广域网中主动队列管理最大的区别是他们可以假设端主机的配合：一个企业的数据中心内，该企业能够同时控制交换机与服务器。这为流类型的区分、流量大小的估计等都提供了极大的便利。然而，在互联网中，我们不能有这样的假设。如果说某算法会对某一类流量进行优先调度，那么所有互联网的用户都会把自己的流量伪装成这一类流量，从而使这一机制失效。事实上，这也正是差分服务[87]等机制在广域网下使用人数较少的原因之一。

相比之下，他们还存在的一个共性问题是假设端上的拥塞控制算法对丢包或者 ECN 标记敏感。然而，随着 Copa、BBR 等新兴的基于速率的或者基于延迟的拥塞控制算法的提出，他们不再对丢包或者 ECN 敏感。因此，如果期望依靠丢包使得端上的拥塞控制算法降低其发送速率，这需要丢包已经十分严重才行。例如，BBR 对于丢包率在 20% 以下的丢包不会做出任何速率上的响应。因此，迫切地需要针对这种延迟敏感的拥塞控制算法设计优化新的主动队列管理机制。

2.3.2　队列大小优化

如何设置瓶颈队列大小一直是网络层管理的一个难题。队列过小会导致网络当中应对突发流量时频繁出现丢包。而队列过大则会导致在速率调整不及时的时候排队延迟过长。因此，设置合适的队列大小一直是网络管理员关心的一个问题，也是降低端到端延迟的一个重要手段。在这个方面上，有一系列经验性的工作进行探讨。例如，2019 年以斯坦福大学教授 Nick McKeown 为首的专家们专门组织了 Buffer Sizing Workshop，来对如何设置交换机队列大小进行讨论。除此之外，还有许多自适应队列大小的工作，例如 ABS[74]等。其会根据网络流量的突发性，来自适应地调整路由器中队列的大小。

另一类队列大小优化上的主要工作就是理论上的分析。最早，学者提出，瓶颈队列的大小应该不少于带宽延时积（Bandwidth-Delay Product，BDP），以使得当时的拥塞控制算法（例如 AIMD 或者 Vegas）至少能够在整个周期内都充分利用链路容量[88]。随后，在 2004 年，Appenzeller et al.[89]提出，其实利用不同拥塞控制流的统计复用特性，瓶颈队列的大小可以降低到 $BDP/\sqrt{N}$，其中 N 为在该交换机上的流的数目。近年来，又有学者指出，随着 BBR 等新型拥塞控制算法的提出，瓶颈队列的大小可以进一步降低到 BDP/N[73]。这样，交换机上可能的延迟的最

大值也会因此不断降低。

然而，这一设置一般只适用于核心骨干网交换机，因为这些交换机通常有上百万条流。在边缘路由器（例如家里的无线路由器）中，可能绝大部分情况下只有数十上百条流。在这种情况下，由于 N 很小，这一结果其实是平凡的。更严重的问题是，在无线网络中，如第1章所述，由于其带宽波动很可能较大，所以队列不得不设置得很长。这其实也直接导致了，很多最后一跳路由器的队列缓冲区都很“深”的情况。在这种情况下，高延迟的发生便是难以避免的。本文工作正是试图在不改变这一设定的条件下，如何缩短端到端的延迟。

2.3.3　端网消息传递

最后一类工作便是在网络层设计新的协议来在端主机和网络设备之间进行更好的消息传递。有良好的消息传递也可以方便地控制延迟，因为在理想情况下，如果端主机能够完美复刻网络可用带宽的变化，便不会有因为自身调整失当带来的拥塞。这方面典型的工作便是 2002 年的 XCP[75]，以及后续的 RCP[76]。他们通过设计新的协议，在协议包头字段包含瓶颈带宽的速率，以精准调控发送速率。这同样也包含一些可能没有设计新协议，但同样在现有协议内捎带了网络状态信息的工作，例如 Kickass[77]和 ABC[78]。Kickass[77]将当前路由器上的某一条流的可用带宽这一信息，通过 IP 分片的大小来传递回发送端。ABC[78]则是利用了差分服务遗留的两个在互联网上未广泛使用的比特（TOS）来标记当前路由器认为这条流需要升速还是降速。

然而，这些工作最大的问题依然是缺乏可部署性的问题。正如在本小节开头所述，网络层的创新层出不穷，但在互联网中能得到真正部署的却是凤毛麟角。这主要原因正是修改网内设备难上加难。上述方案都需要同时修改网内设备以及端主机设备。这在实际场景中便十分困难：端主机设备通常是内容提供商来维护（如：百度、阿里等）；而网内设备则是设备制造商来维护（如：华为、华三等）。同时协调两方进行修改以实现性能收益，在互联网发展的历史长河中，早已被证明十分困难。

在本文的设计中，我们始终恪守尽量减少对设备的修改这一原则。提出的工作均能够在只修改单个网络设备的情况下进行部署取得收益，不需要与其他设备进行交流协同。这样，本工作便具有一定的可部署性，同时也有了一些在现网实际部署的例子。

2.4　本章小结

本章从实时多媒体的特性以及低延迟网络的特性出发，按照现有的互联网体系结构，分别从应用层、传输层、网络层的角度介绍了在过去一段时间内，学术界和工业界在这两方面优化上做出的努力。本章首先介绍了应用层上在实时编解码器、自适应码率算法、多媒体传输协议上的设计工作，紧接着介绍了传输层上在拥塞控制和丢包恢复上的相关工作，最后介绍了在网络层上从主动队列管理，到路由器队列大小管理，再到端网协同优化上的工作。在介绍的过程中，本章还分析了现有工作的不足，为后面介绍本文工作进行了铺垫。

第 3 章　实时多媒体传输架构

如第1章中所述，当实时多媒体传输应用的优化目标转向尾延迟时，延迟的主要来源可能与原有架构中对中位数、平均延迟等分析的来源不再一致。在本章中，我们着重分析，在现有的实时多媒体传输架构下，端到端延迟的波动到底源自哪里。本章将首先在总体上对延迟波动产生的来源进行分析，接下来从控制通路和数据通路两个角度分析延迟波动产生的原因。

3.1　延迟波动来源分析

在传统的实时多媒体传输中，延迟的主要组成部分就是网络延迟 — 当发送端和接收端的物理距离仍较远、拥塞控制等机制仍会产生很长的队列时，网络延迟便会占据大部分延迟。然而，如第1章所述，随着边缘节点的下沉部署、网络拥塞控制等机制的改进，网络延迟已经不再是延迟的主要组成部分。当前的现实情况是，在很多实时多媒体传输应用中，应用服务提供商会通过大量投资来换取平均或者中位数延迟甚至都可以低到 10-15ms。例如，像云游戏这一类应用，服务提供商会将服务器从原来可能全国的几个节点，下沉部署到每个省份、每个地区都有若干节点。在这种情况下，针对绝大部分用户，在用户所在的城市就很有可能会有一个计算节点来为用户提供服务。这样，数据包只需要在城域网之内传递，极大缩短了网络延迟。类似地，当接入网技术不断从 4G 升级到 5G，WiFi 4 升级到 WiFi 6 后，最后一跳接入网的无线链路传输延迟也得到了很大程度的改善。

然而，当实时多媒体传输的关注点转向千分之一、万分之一的尾部延迟时，任何一个环节的微小波动均可能会导致端到端延迟在第 99.99 百分位上升。现有的实时多媒体传输架构在设计之时确实有考虑适应不同网络情况下的波动，但对网络从一个状态到另一个状态的过渡、收敛过程关注的还不够。这也是自然的 — 当关注的延迟百分位在 50 分位乃至 90 分位时，其实并不需要怎么关心这些瞬态的收敛过程。然而，当应用的目光放到这些在很靠后的尾部延迟的时候，这些瞬态的收敛过程便十分关键了。因此，本节主要分析当实时多媒体传输关注这些千分之一的截止时间错失率、尾延迟之后，延迟波动的可能来源。

本工作发现的一个重要的延迟波动来源便是控制通路延迟的存在。正如前所述，网络状态是时时刻刻都在波动的，因此端上的响应需要时刻根据网络状态来进行调整。然而，由于**控制回环**的存在，端上的响应往往都是滞后的。形式化地来说，在 t 时刻端上的响应动作 $a(t)$ 并不能基于 t 时刻的网络状态 $s(t)$，而是基于

$t-\tau_{control}$ 时刻的网络状态 $s\left(t-\tau_{control}\right)$，其中 $\tau_{control}$ 便是控制通路的延迟。因此，当网络状态发生变化时，端上的响应动作 $a(t)$ 往往会滞后于网络状态的变化，从而导致端到端延迟的波动。

这在当应用关注点从延迟转向尾延迟时，就变得非常重要了。在之前，当网络状态发生变化的时候，诚然，端上可能做出的响应稍晚了些。但只要端上能够做出正确的反应，发送速率等参数便可以收敛到新的稳态值。而这一瞬态的过程往往是短暂的，因此并不会对中位数、90 分位等延迟造成影响。但网络波动确实是会偶尔发生的。例如，第4章的一个测量表明，在部分无线网络的真实数据下，网络带宽降低为原来 50 分之一的概率可能高达 1%。在这种情况下，控制通路的延迟便十分关键了。

同时，数据通路中不同组成部分在端到端延迟中的角色也会有所变化。随着边缘数据中心的部署、5G 和 WiFi 6 等新兴接入网技术的出现，网络端到端延迟在中位数（一般情况下）可以甚至做到 10-20 毫秒[90]。在这种情况下，当我们观察到某一视频帧的延迟上升到数百毫秒时，其可能的造成原因便不再仅仅是因为双方的物理距离过远了。应用层、传输层、网络层的延迟波动都有可能会导致瞬间的延迟上升。

本章余下的两节将会对这两个方面进行分析。图3.1展示了我们建模分析后的实时多媒体传输架构中可能对延迟抖动产生影响的一些组成部分。在控制通路上，反馈延迟和决策延迟均会影响控制通路本身的延迟。在数据通路上，应用层、传输层、以及网络层的延迟也都会影响最终数据通路上的端到端数据延迟。本文中涉及的几项工作也是按照这两个方面分别进行展开，力求系统地解决实时多媒体传输中延迟波动的问题。

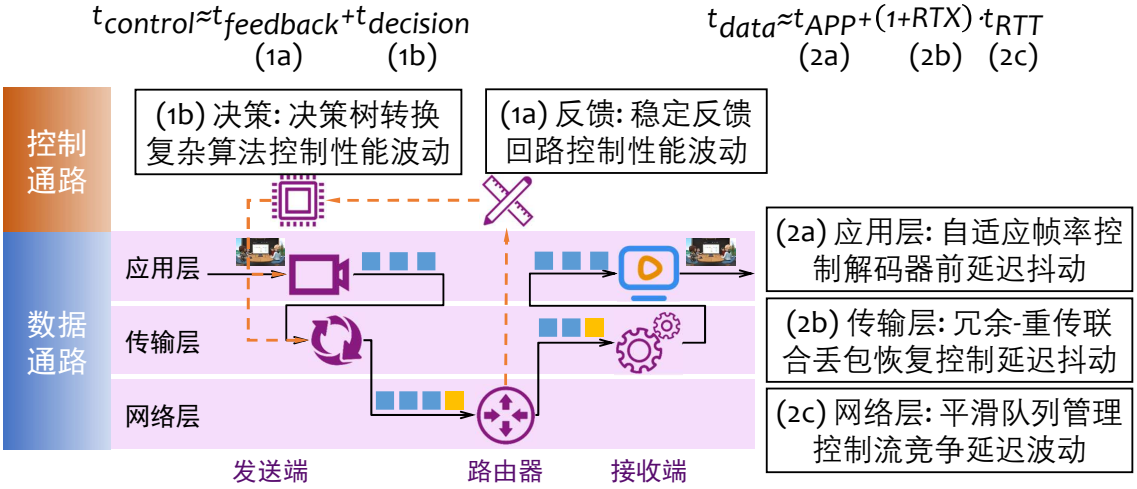


图 3.1　实时多媒体传输体系结构及本文几项工作的关系

3.2　控制通路延迟

本节首先识别定位了**控制通路**对在尾部端到端延迟出现波动的重要作用。控制通路对实时多媒体传输的延迟影响是间接的，况且仅仅在应用关注尾部延迟时才会有作用：当网络状况发生了变化时，如果多媒体传输的发送端调整自己的发送速率**慢了些**，这便有可能导致延迟波动带来性能的下降。请不要小瞧这一点点响应的时间：如果每次网络状况变化时，发送端都需要数百毫秒来响应，那么这数百秒中多媒体传输的用户体验便是不好的。然而，互联网总是波动的 – 如果每几分钟网络状态便有些许波动，这就意味着用户会有千分之几的概率遇到延迟波动带来的性能下降。这种情况下，控制通路的延迟就变得非常重要了。

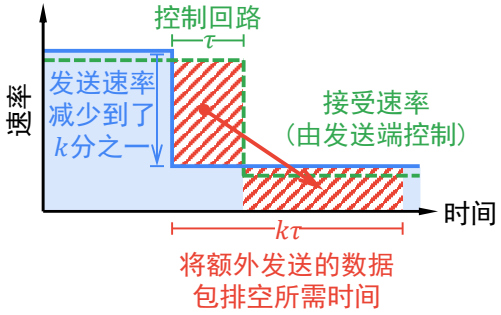


图 3.2　一个在可用带宽下降时控制通路延迟的例子

图3.2展示了一个说明性的例子。在互联网中，一条流的可用带宽可能会因为无线信道的干扰、竞争流量模式的改变而时刻产生波动。这时候，实时多媒体传输的发送端的发送速率要随之进行改变，以让自己的吞吐率实时适配当前的可用带宽。不失一般性地，当一个实时多媒体流在瓶颈路由器（实线）的可用带宽突然下降为原来的 k 分之一时，发送端的发送速率也需要尽快进行降低，以适配新的可用带宽。然而，如前所述，可用带宽的下降并非可以立刻被发送端知晓，而是需要一个控制通路延迟 τ（即控制回路）才能最终反映到发送速率的下降上。在这种情况下，发送端发送速率的减少便会被可用带宽的减少向右偏移一些，如图中虚线所示。图3.2中的实线是瓶颈路由器的可用带宽，虚线是瓶颈路由器的发送速率，红色阴影是瓶颈队列的积压。在这段时间内，瓶颈队列仍然以原始发送速率接收数据包，而其处理速率却因可用带宽的下降而大大减少。因此，这些过多的数据包会导致瓶颈队列的积压，如红色阴影所示。

更糟糕的是，当控制通路延迟为 τ 时，用户体验到延迟恶化的时间却很可能远远不止 τ。这正是本文的另外一个关于控制通路延迟很重要的观察 — 当网络状况发生波动时，由于事实上网络的可用带宽是在变差了，因此这些超出网络承载能力的数据包需要数倍于原来堆积起来的时间才能排空。在这里也结合图3.2的这个例子进一步分析一下。具体来说，到达瓶颈队列的数据包在控制循环 τ 期间需

要 k 倍的时间（$k\tau$）才能发送出去。这是因为，可用带宽下降前发送的数据率远大于下降后的新的可用带宽。因此，原来可能 1 个时间单位堆积起来的数据，需要 k 个时间单位才能缓解掉。这就像图中所展示的那样，图中事实上两块红色阴影的区域面积是相等的。在这段时间内，所有发送出去的数据包都会经历增加的延迟，从而降低用户的体验。

具体来说，控制通路的延迟又分为两部分，即反馈延迟和决策延迟：

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

其中，反馈延迟 $t_{feedback}$ 是指从发送端到接收端的反馈信号的传输时间，决策延迟 $t_{decision}$ 是指发送端根据反馈信号做出决策的时间。而这两个环节的抖动均可能会导致最终的端到端延迟的波动。本节首先介绍这两部分的功能，其次讨论它们是如何影响最终的端到端延迟波动的。

1. 反馈延迟抖动导致的端到端性能波动。如何获取信息的反馈是几乎所有控制系统都会面临的重要问题。从电路、信号到实时多媒体传输中的编码调整、容错调整、发送速率调整等等，这些都离不开反馈在决策中的重要作用。这在网络领域诸多对现有工作的分析中也有所体现。例如，QCN[91]在分析稳定性时，假设网络内交换机发送的 QCN 信号要经过 τ 时间后，才能被发送端获取到。因此，这种反馈延迟事实上是普遍存在的。

端到端控制算法依赖于对网络状态的及时获取。例如，TCP 拥塞控制算法会依据数据包的延迟、是否丢包以及一段时间内的速率变化来判断网络的拥塞程度。当网络状态出现变化时，这一变化会立刻反映到数据包的延迟、吞吐、丢包率等指标上。然而，我们发现，在网络状态发生抖动时，这些指标往往不能够立刻被发送端获知。在这一瞬间过程中，发送端发送速率与网络状态的不匹配，便会导致端到端性能的波动。在这里，现有工作有如下两个方面的缺陷：

第一，现有工作一般假设反馈延迟是不变的。这样可以极大地简化许多建模与分析。然而，本文第4章的一个重要观察是，在尾部的情况下，反馈延迟事实上是会随着数据通路延迟的膨胀而一同膨胀的。这是因为，在网络层以上，控制信息并没有单独的控制通路，而是也要通过数据通路进行传递。因此，如果数据通路由于数据包排队等原因造成了延迟的膨胀，那么反馈延迟也是会随之一同膨胀的。这就会导致发送端知晓网络状态变化会更晚些，从而进一步恶化端到端延迟。

第二，现有工作对反馈延迟的分析更多在于稳定性而非性能。反馈回环的存在对许多稳定性分析都非常重要：通常来讲，如果反馈回环过长，以至于比网络状态变化的周期还要长，那么这个控制系统很有可能是不收敛的。但在当前的互联网中，反馈回环一般远小于网络状态的变化。例如，一般反馈回环大约也就是一

个往返时延，这在一般的实时多媒体传输应用中大约为几十毫秒。而网络状态一般不会每几十毫秒就发生什么剧烈变化，所以通常的分析结果都是稳定的[91]。然而，当实时多媒体传输对延迟的要求关注到第 99.9 百分位甚至更靠后时，收敛过程这点延迟波动也会对用户的性能产生影响。

第4章对此进行了详细分析，并提出了通过解耦数据通路和控制通路来稳定反馈延迟的机制。如图3.1所示，第4章的工作将主要优化控制通路的反馈延迟 $t_{feedback}$。其主要做法为，不再像传统协议那样将控制信息搭载在原有的数据报文中，而是解耦控制信息与数据报文。这样，无论数据通路延迟如何波动、膨胀，控制通路的延迟仍然能够保持相对稳定。在这种情况下，发送端总能相对及时地获知当前的网络状态，并做出相应调整。

2. 决策不稳定导致速率调整滞后或出错。获取网络状态信息后，如何决策则是实时多媒体传输控制算法紧接着面临的另一个重要问题。网络中的信号往往是充满噪声与歧义的。例如，当发送端观测到一个丢包事件时，这有可能代表发送端需要降低发送速率以缓解网络拥塞，也可能仅仅代表无线链路的信道质量下降而事实上不需要发送端降低发送速率[92]。因此，决策算法一般倾向于只当收集到足够多的信息以有足够的信心来做出决策时才做出决策，这样决策才足够保靠。

近年来，由于应用对性能的不断追求，拥塞控制、视频码率调整等控制算法的决策逻辑已经越来越复杂。从传统的寥寥数行的启发式方法开始，研究者们逐渐转向使用神经网络[23,93]或整数规划[38]等方法来进行决策。这些尝试是有益的：一方面，研究者们对网络链路的了解越来越深入，他们能够有能力去针对性地对网络做出一些假设和建模。另一方面，受神经网络等新兴机器学习技术在计算机视觉、自然语言处理等领域的进展所鼓舞，研究者们也倾向于相信这些神经网络在实时多媒体传输相关的领域内也大有可为。然而，现有的决策算法在两个维度上目前面临一些问题：

第一，决策算法的决策延迟日趋增加。采用神经网络或者整数规划这类算法最直接的缺陷就是其极高的决策延迟。传统的数行代码即可实现的启发式算法在实际线上部署运行时几乎不会消耗多长决策时间。然而，神经网络即使在前向传播的推理阶段也会消耗较多的计算资源。例如，传统的拥塞控制一般每收到一个数据包就对拥塞窗口进行更新。在一个吞吐量为 30Mbps 的流中（这是云游戏等高清实时多媒体的常见流量大小），这意味着两个数据包的间隔最长为 0.4 毫秒。然而，神经网络的前向推测即使采用了 GPU 等专有加速硬件的情况下，依然可能会消耗毫秒级的延迟。这个现象在整数规划等优化求解的建模方法中更为严重。复杂的整数规划可能需要数分钟乃至数小时才能求解出来。这便造成了一个决策延

迟高的现状与网络算法要求高频决策的需求之间的矛盾。

第二，这些算法的决策逻辑也越来越黑盒化，网络管理员很难了解其决策背后的逻辑。例如，神经网络通常包含成千上万（有时候甚至上亿[94]）个神经元，通过复杂的运算方能输出其决策的结果。管理员一般很难理解一个决策是如何做出的。这就导致了一个可能可怕的事实 – 决策可能会出错而不被发现，这让网络管理员如何信任这一模型？这在整数规划等优化算法中也是类似的。当求解的结果严重偏离网络管理员的常识时，管理员并没有办法知道是哪一个约束或者哪一个变量的设计出现了问题。这种情况下，错误的决策同样会导致端到端性能的波动。

第5章对上述两个问题也进行了分析，并提出了能够通过将复杂算法转换为轻量、稳定的决策树的算法来避免因为决策不稳定导致的端到端性能波动。如图3.1所示，第5章的工作将主要优化控制通路的反馈延迟 $t_{decision}$。其主要做法为，将线下优化与线上部署解耦开，不再像现有工作一样将线下优化与线上部署绑定在一起。网络管理员在线下训练时，依然可以采用其认为性能高、效果好的算法及模型进行优化。但当将这一优化好的模型进行线上部署运行时，该方法能够将其在性能损失很低的情况下转化为既具有低决策延迟又具有可解释性的决策树模型。这样，决策部分的延迟和可靠性在绝大多数情况下便可以得到保障。

3.3　数据通路延迟

本节首先定性地分析一下数据通路的端到端延迟的组成部分。在非常理想的情况下，数据通路的延迟会受到以下几个因素的影响：

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

其中，t_{app}、RTX、和 t_{RTT} 分别是应用层的处理时间、重传次数和往返时延，他们事实上对应着数据通路中应用层、传输层、网络层对端到端延迟的影响。具体来说，我们首先介绍下这三层的（可能的）设计缺陷会如何影响数据通路端到端延迟。

- t_{app} 是应用层的处理时间，其主要和应用层的设计有关 – 例如，如果应用层需要对视频帧进行编码，那么编码时间便会增加。
- RTX 是重传次数，其主要和传输层的丢包恢复设计有关 – 例如，如果一个包丢了 RTX 次，其便会在第 $1 + RTX$ 次传输时到达接收端。
- t_{RTT} 是往返时延，其主要和网络层的队列管理设计有关 – 例如，网络内的队列越长，往返时延便会越长。

例如，假设某数据包在发送端的编码器首先处理了 5ms（应用层）。然后，该数据

包在传输层开始准备进行传输。当前的网络 RTT 是 30ms。不幸的是，这个数据包在网络中总是被丢弃，直到第 4 次传输才成功到达接收端，被接收端确认。假设这里在最理想的情况下，发送端总能在 1 个 RTT 内就判断到这一个数据包被丢弃了。因此，在传输层，这个数据包总耗时为 30×4=120ms。到了接收端之后，其在应用层可能还会有一些附加的延迟。例如，视频的解码可能就会耗时 10ms。这样，按照公式3.2的估计，总的端到端延迟便大约是 15ms+120ms=135ms。

这个模型当然存在一些近似之处。例如，在事实上，发送端不见得能在第一时间发现数据包被丢弃。事实上，TCP 在数据包第一次丢弃后，需要等待后续 3 个数据包都成功到达并被确认才会触发快速恢复机制。在第二次丢弃后，则要等到重传超时（RTO）后才会继续重传。上述公式只是估计的一个理想情况 — 我们总能通过改善协议设计来达到估算中的理想情况，例如 WebRTC 框架中的 RTP/RTCP 的 NACK 设计。但总之，上述估计给出了一个关于数据通路延迟的主要组成部分的分析。

然而，近年来，在新一代多媒体传输中，这几部分的情形均各自发生了一些变化。这三个环节的抖动也均会导致最终的端到端延迟的波动。

1. 应用层延迟：视频质量增加导致应用层编解码与网络协同过程中出现波动。在应用层延迟中，本文注意到的一个比较大的变化是在应用与协议栈接口处（例如，套接字缓冲区）中的等待延迟。现有应用层的设计并没有考虑到应用出现瓶颈可能引入的延迟波动 — 当前操作系统中套接字的缓冲区只会被动等待应用向其读取数据，而不会进行主动队列管理。当应用能够来得及处理当前网络发送过来的数据时，应用会主动从缓冲区中读取数据。当应用暂时来不及处理这些数据的时候，这些数据便会堆积在缓冲区中等待。当缓冲区逐渐减小时，当前的 TCP 协议会相应修改流控窗口（advertised window），来告知发送端减少发送的数据量。当缓冲区被填满后，接收端便不再接收网络中的新数据，直到应用处理了一部分数据腾出来空间为止。

然而，这样的设计面临的一个直观的问题就是，如果应用处理不及时，队列会持续堆积直到堆满。这就好比在网络中路由器上，队列如果不主动管理就会堆满直至溢出一样，在这种情况下，就会产生很高的排队延迟。这样的设计对包含实时多媒体传输在内的低延迟的应用是很不友好的。

这个队列的问题随着多媒体传输应用的发展而越来越严重。随着新一代多媒体对画质的要求逐渐变高，视频编解码的计算量也随之增加。例如，分辨率从过去的 240p 发展到如今的 1080p，未来可能还会出现 4K、8K 等更高分辨率的实时多媒体传输。视频的帧率也从视频通话的约 24fps 增长到 60fps、90fps 甚至更高。这

都会让应用处理数据的负载越来越重，进而导致端到端延迟的波动。

基于此，第6章对上述问题进行了详细的分析，并提出了一种应用层的队列管理机制，能够在应用出现瓶颈时主动控制应用前面的队列，从而避免端到端延迟的波动。尤其是在当下越来越普遍的高清晰度、高帧率的实时多媒体传输中，采用第6章提出的机制将越来越迫切。如图3.1所示，第6章的工作将主要优化数据通路的应用层延迟 t_{app}。其主要做法为，通过对这一缓冲区队列进行主动管理，而不是等待其被动溢出。就像路由器上的主动队列管理算法一样，当缓冲区队列开始增长时，通过应用层的协议通知发送端降低发送速率，以避免缓冲区堆积到较高的位置。这样，便可以把应用层的延迟有效地控制下来，让端到端延迟波动的情况更少些。

2. 传输层延迟：对延迟波动要求的提升使得现有传输层丢包恢复机制无法满足。在传输层延迟中，本文注意到当对延迟的百分位的关注从 50 分位、90 分位提升到 99.9 分位、99.99 分位时，现有的传输层丢包恢复机制已经无法满足这一要求。当前传输层很多设计都没有考虑那些小概率尾部情况可能带来的时延问题。一个典型的例子就是丢包恢复。当数据包没有被丢失时，在延迟上大家相安无事。然而，如果一个数据包不幸丢失了，传输层现有的设计便是在第一次时候触发快速恢复机制，但第二次可能要等待一秒钟才能触发超时重传。尽管有许多设计尝试加速这一过程（例如，TLP[54]），但重传仍然通常是不可避免的。这样的一个问题是，当真的比较极端的情况发生的时候，在这些极端情况下的数据包的延迟可能会非常差。例如，如果当网络的瞬间丢包率达到 10% 时，一个数据包需要被传输 4 次才能到达接收端，那么这个数据包的延迟会增加 4 倍。而在丢包率为 10% 的时候，一个数据包被传输 4 次（即连续被丢弃 3 次）的概率也高达千分之一。这便会直接影响用户在尾部的体验。

需要注意的是，尤其是针对实时多媒体传输的视频帧而言，对于一般的解码器，只有当一帧的所有数据包都到达接收端后，这一帧才能被交付给应用进行解码、渲染。也就是说假设一个视频帧可能有 50 个数据包，那么哪怕有 1 个数据包出现了上述的情况，那么这一个视频帧便会让用户的体验因之下降。因此，现有的丢包恢复的机制很难满足新一代多媒体传输对延迟抖动的极致要求。

这个尾部的问题也随着实时多媒体传输应用的发展而变得越来越严重。游戏用户可能可以容忍千分之一的卡顿率，但像是远程手术、远程辅助驾驶等应用其实会需要万分之一甚至更低的卡顿率。想象一个复杂手术或者一段长途旅行可能持续十数个小时，其中哪怕卡顿了几秒钟都可能是非常致命的。这十数个小时中的几秒钟就大概是万分之一甚至更低的卡顿率要求。同时，随着视频传输码率的

提高，但网络中的数据单元（Maximum Transmission Unit，MTU）并没有随之提高，前述的问题中的一个视频帧包含多个数据包、被其中一个数据包拖慢的现象会更加严重。因此，对以丢包恢复为主的这些在设计时并未良好考虑尾部延迟波动的应用的重新考虑便十分重要了。

基于此，第7章同样对上述问题以测量数据为支撑进行了详细的分析，并提出了一种传输层的丢包恢复机制。该机制能够在带宽成本甚至有所降低的情况下，大幅缓解因为丢包重传而带来的实时多媒体传输端到端延迟抖动。尤其是在对尾部的追求越来越极致时，采用第7章提出的机制将越来越迫切。如图3.1所示，第7章的工作将主要优化数据通路的传输层延迟，即其重传次数 RTX。其主要做法为，结合两种常见的丢包恢复机制（冗余编码与丢包重传），让传输层在进入尾部情况越来越可能的情况下（例如，第 2 次、第 3 次传输）加大冗余编码的力度，来避免出现极端恶劣的情况。这样，便可以有效控制因为小概率事件的发生而导致的端到端延迟波动。

3. 网络层延迟：拥塞控制算法的多样化使得网络层队列管理机制出现性能波动。本文注意到，近年来，网络层的队列管理机制所面临的流量特征和应用需求也发生了一些变化。如前所述，网络层中延迟的主要来源就是排队。而排队的造成一般即是由队列的到达速率与发送速率不匹配带来的。在传统的网络层的队列管理中，有 CoDel 等算法[68]通过对队列的长度进行限制，来避免队列过长导致的延迟波动，并得到了广泛的部署。然而，这些算法在当今的网络传输的流量特征中面临下面两个问题：

第一，队列管理算法的响应更多针对丢包而不是其他指标。当前的队列管理算法，在设计之初，针对的是一些较为传统的 TCP 拥塞控制算法，例如 Reno[55]、CUBIC[56]等等。这些算法最突出的特点，就是他们高度依赖于丢包信号（或者 ECN 信号）来调整速率。比方说，当发送端没有观测到丢包时，它便会持续尝试去增加发送速率（或拥塞窗口）。当发送端观测到了丢包时，它便会减少发送速率。然而，以实时多媒体传输应用为代表的低延迟应用其实不再采用类似 Cubic 这种丢包敏感的拥塞控制算法，而是采用类似 GCC 这种延迟敏感的拥塞控制算法，以获取低延迟。这就导致现有的队列管理算法可能不见得能有效控制延迟：延迟敏感的拥塞控制算法不再对丢包信号响应。这便让低延迟的确保变得不那么简单。

第二，队列管理算法的设计更多关注于宏观表现而不关注微观表现。当前队列管理算法在衡量性能时更多关注的是长时间尺度上的宏观表现。例如，在公平性上，研究者们会去测量在比较长的时间尺度上的吞吐量公平性指标（例如，Jain's 公平性指数）。但对在瞬态下如何慢慢收敛达到这一公平性并没有太多关注。在此，

当性能关注的重点转移到尾延迟时，如前所述，这一收敛过程的瞬态性能便同样很关键了。尤其是当网络上的一些竞争流量（例如，网页浏览）也出现了一些新的变化时，网络层的队列管理机制就难以保障多媒体传输的延迟波动在可接受的范围之内。如果，像现有方法一样，并不关注在瞬态下性能的波动，那么用户便会在尾部遭受很差的体验。

基于此，第8章对队列管理算法的瞬态性能与拥塞控制算法的响应信号进行了分析，提出了一种网络层内的主动队列管理机制，通过限制突发流量对稳定流量的瞬时干扰来管控性能波动。我们发现，随着网页设计的越来越复杂、应用的性能需求越来越多元，第8章的机制也将发挥越来越大的作用。如图3.1所示，第8章的工作将主要优化数据通路的网络层延迟 t_{net}。该工作的主要做法为，一方面不再依赖于丢包信号来向拥塞控制算法传递信息，而是使用延迟；另一方面在瞬态的状态转移上采用平滑的转移策略，让拥塞控制算法有充足的时间响应网络状态的变化。这样，队列管理机制可以有效地控制在网络流量等出现波动的时候，控制延迟的波动。

3.4　本章小结

本章详细分析了实时多媒体传输应用的架构以及影响端到端延迟的组成部分。本章按照本文的控制通路和数据通路的结构划分，分别介绍了控制通路中反馈和决策可能对用户延迟带来的波动，以及数据通路中应用层、传输层和网络层中的组成部分延迟波动带来的用户延迟的波动。在具体每个组成部分中，本章又简要介绍了当前主要存在的问题，以及本文即将提出的解决方案。

第 4 章　控制通路反馈：通过拥塞早反馈优化反馈延迟

4.1　本章引言

传统无线链路上的拥塞是由用户可用带宽突然下降引起的，例如多用户访问和相互干扰。无线网络的可用带宽在第 99 分位数上可能下降 10 倍（§4.2.3）。在这样的带宽骤降之后，数据包很快在无线接入点上开始排队，从而导致端到端延迟的增加。理想情况下，发送方在带宽减少时能够快速做出反应，例如通过降低比特率来防止排队积压、高延迟和丢包。不幸的是，我们发现发送方在这种情况下**从根本上是非常有限的**。更糟糕的是，队列积压时发送方反应事实上是最慢的。

原因在于，拥塞信号是沿着同样的拥塞路径传输的。简单来说，为了观察瓶颈队列是否正在填充，发送方必须首先收到一个实际**等待在该队列中**的数据包的确认。因此，在发送方需要像时间戳或丢包这样的拥塞信号时，这些信号需要更长的时间才能到达发送方。在图4.1中，我们展示了数据包和它们携带的控制信号（如时间戳或 RTT）传输的路径。

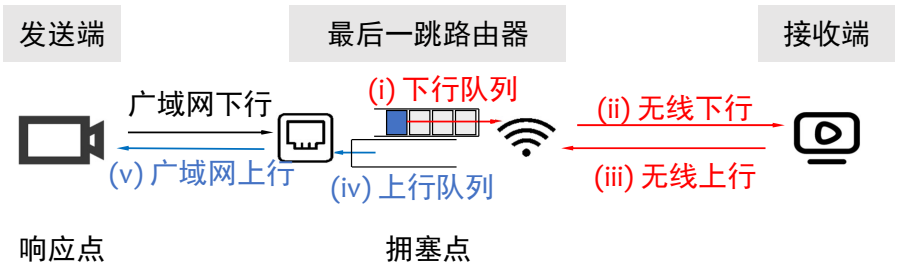


图 4.1　无线最后一跳上的控制回路

本章工作的关键发现是，控制回路可以从数据包需要走过的完整路径中解耦出来，从而保护控制信号不受堆积起来的、可能很长的[45]队列的延迟的影响。当观察到下行队列正在填充时（图4.1中的 `i`），一个精心设计的无线接入点可以修改或延迟上行队列中的反馈确认包（图4.1中的 `iv`），从而允许拥塞信号在不受瓶颈延迟的影响下到达发送方。因此，简而言之，相比于现有方案，本工作希望绕过段 (i)-(iii) 以实现最短的控制回路。

有很多研究工作试图改善无线网络的网络延迟，但这些方法主要是为了改善实时多媒体应用程序在无线网络中的**中位数**而不是**尾部**延迟。我们认为问题主要源于这些方法都依赖于由于拥塞信号需要穿过拥塞的高延迟路径而产生的滞后的控制回路。例如，端到端解决方案（如拥塞控制算法）在发送方收集端到端信号（例如每个数据包的延迟）以调整发送速率。但是，仍然需要通过一个（在这里事

实上已经膨胀了的）控制回路以便在发送数据包后收集信号。同样，网络内解决方案（如主动队列管理）能够主动标记一些信号（例如数据包丢失），但这些信号仍然需要通过接收器向发送器反弹，这又导致了一个长的控制回路。

尽管本工作的关键发现很简单，但在实践中实现它是具有挑战性的：

（1）接入点如何在数据包尚未传输时预测数据包延迟？直觉上，接入点可能只需测量下行队列中排队的字节数并除以可用的链路容量来测量排队延迟。但是，回想一下，链路带宽是波动的（这正是本工作要解决的场景），因此这样的估计可能并不准确。

（2）接入点如何将这个信息以一种可部署的方式传回给发送方？一个直接的解决方案是让路由器直接定义一种新的消息格式，并把这种消息传回给发送方（例如 XCP[75]或主动网络[95]）。但是，同时修改路由器和发送端会为大规模部署产品（§4.2.3）建立部署障碍，因为路由器和发送端通常由不同的组织维护。此外，对于现有的部署协议，有些使用显式信号（例如时间戳），有些使用隐式或带外信号（例如 RTT 或 RTT 梯度）。有些协议对 RTT 的加权移动平均值感兴趣[57]；有些协议对特定窗口内的最小 RTT 值感兴趣[49]；有些协议对数据包之间的时间间隔感兴趣，而不关心 RTT[60]。接入点必须以一种普遍适用于这些不同协议的方式来修改或延迟上行数据包，以便发送方和接收方都不需要修改就可以部署。

为解决以上挑战，本章提出了 Zhuge①，该方法通过最小化控制循环来在无线环境中实现一致的低延迟②。Zhuge 包括一个“命运预测器”模块，该模块在下行队列中到达数据包时对该数据包的延迟进行预测，该延迟将到达接收器并返回到无线接入点。命运预测器分别估计了两个因素（§4.4.1）对排队延迟的影响，并使用这些因素为每个到达的数据包分别估计不同延迟再综合评判。Zhuge 的第二个组件是“反馈更新器”，它修改上行数据包。根据协议类型的不同，它可基于命运预测器来修改反馈包记录的原始数据包延迟，也可以控制确认包之间的延迟的差值（§4.5.2）。

本章在仿真和 WiFi 路由器测试床上实现了 Zhuge（§4.7）。使用真实世界的无线数据集和配置的评估结果表明，Zhuge 可以将网络条件（例如尾延迟）和应用程序性能（例如视频帧延迟）的关键指标提高 17% 到 95%。进一步的评估还表明，Zhuge 能够在不同的场景下在现实世界中获得令人满意的性能。

① Zhuge（诸葛亮）是中国古代一位预测很准的军师，人称“神机妙算诸葛亮”。

② 我们主要关注最近设计的拥塞控制算法，这些拥塞控制算法旨在保持低延迟，但无法一致地在尾部也实现低延迟。尝试去填充瓶颈队列的拥塞控制算法（例如 CUBIC[56]）一直都有高 RTT，这不是我们的目标。

4.2　背景与动机

本节首先利用在真实世界测量的数据来浅析无线网络尾延迟的现状 (§4.2.1)，接下来讨论现有方法在保障可靠低延迟上的不足 (§4.2.2)。最终，本节提出通过缩短控制回路来优化尾延迟的动机 (§4.2.3)。

4.2.1　理解无线尾延迟

我们首先回答一个问题——为什么无线延迟会在尾部出现波动？

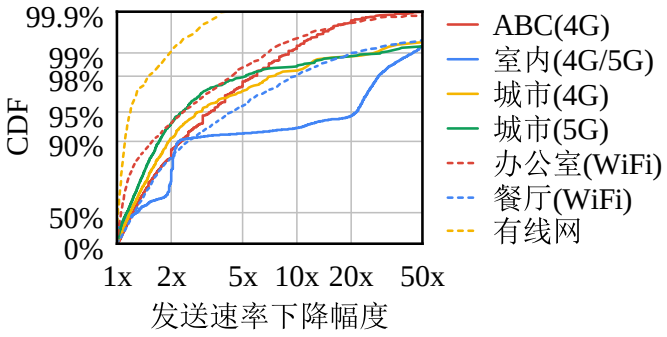


图 4.2　无线网络可用带宽下降幅度的累积分布

突出的尾延迟是由于发送速率和瓶颈队列的可用带宽不匹配所导致的。如第3.2节中所分析，延迟的瞬间上涨取决于 (1) 可用带宽的波动有多剧烈 (k)，以及 (2) 发送端多快能够反应过来 (τ)。对于可用带宽波动 k 而言，无线信道天然就比有线信道更加波动。可用带宽的突然下降会导致延迟的瞬间上涨。

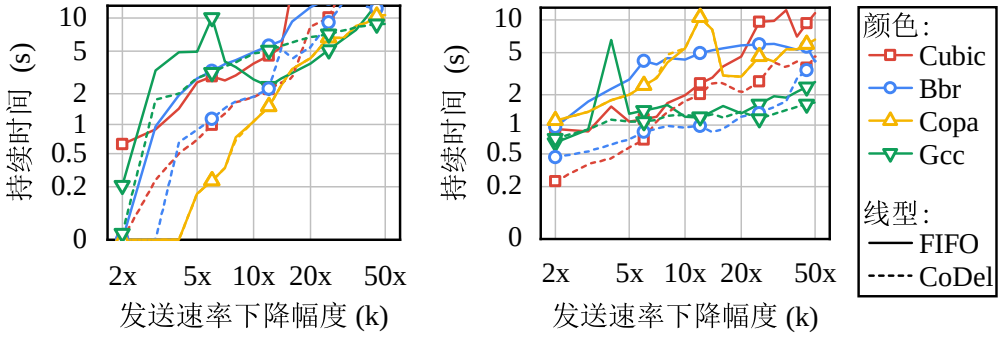
我们计算了若干个数据集每隔 200 毫秒的平均可用带宽，如图4.2所示。图4.2中的实线是开源数据集[78,96-97]，虚线是我们在测试平台上测量的。可用带宽的测量值是切分成 200ms 窗口后每个窗口的平均值。考虑到常见的互联网 RTT（几十毫秒），拥塞控制算法在这样一个时间尺度下应该对波动有所反应。数据来源包含公开数据集和自己在办公园区及餐厅的测量，细节详见 §4.7.2。

如图4.2所示，对包含 5G 毫米波及 5GHz 频段的 WiFi 在内的所有无线数据集，有 0.6-7.3% 的可用带宽波动幅度大于 10×。这远大于有线网络 <0.1% 的测量结果。对于控制回路 τ 而言，在大多数情况下，拥塞控制算法需要至少一个 RTT 来接收到拥塞相关的信号（如：端到端延迟增加、数据包丢失等）进而调整发送速率。当瓶颈处队列开始堆积时，端到端的 RTT 同样会因此膨胀，进一步减缓了拥塞信号回到发送端的过程。因此，端到端延迟在尾部会出现波动。

4.2.2　现有解决办法

可用带宽的下降幅度 k 是没有办法控制的，因为它是由于链路层和以下的内容（如：无线干扰）所导致的。因此，我们将研究的重点放在了如何减少控制回路 τ 上。许多传输层的创新都是为了改善一个连接的稳态情况下的中位数延迟。例如，BBR[57]将拥塞控制的工作点从 CUBIC[56]中的满队列移动到了空队列。CoDel[68]队列管理也试图在各种网络条件下缩短原来 FIFO 队列的稳态长度。后续的研究工作（包括拥塞控制[49-50,60]和主动队列管理[98]）进一步提供了维持不同反馈信号下的最佳工作点的有用思路。当我们将这些工作综合应用起来，站在巨人的肩膀上，应用程序的中位数延迟便可以很好地控制。然而，这些方法都是在稳态下提升性能的，对控制尾延迟而言，它们是不够的。

第一类工作是基于端主机的解决方案。对于网络层及以上的内容，现有的基于端主机的解决方案无法快速地适应可用带宽的减少，因为它们的控制回路过长和过于复杂。回顾图 4.1，当绿色阴影的数据包到达拥塞点并观察到长队列时，它首先需要通过队列（i），传输到接收端（ii），接收端的反馈传输到接入点（iii），最后发送到发送端（iv 和 v）。由于发送端被通知的最短时间是一个完整的控制回路，包括段（i）-（v），一个纯端到端的拥塞控制算法无法及时地适应瞬态带宽波动。我们进一步模拟了最近的延迟敏感的拥塞控制算法（BBR[57]，Copa[49]和 GCC[60]）以及主动队列管理算法的性能，如图 4.3所示。RTT 恶化持续时间是 RTT>200ms 的时间。拥塞窗口（或发送速率）的减少持续时间是拥塞控制算法重新收敛的时间。当可用带宽减少 10× 或更多时，所有的这些算法，无论是否与延迟敏感的主动队列管理算法一起使用，都会遭受几秒钟的 RTT 恶化。如前所述，膨胀的 RTT 会导致更严重的拥塞，进而导致更长的 RTT 延迟。



(a) RTT 的恶化持续时间　　(b) 拥塞窗口（或发送速率）减少持续时间

图 4.3　无线带宽下降后不同拥塞控制算法和主动队列管理机制的收敛时间

第二类工作是网内解决方案。哪怕是基于网内优化的解决方案也无法及时地反馈这些信号。例如，主动队列管理算法如 CoDel[68]会在队列的前端丢弃数据包，

以减少下行队列延迟（图 4.1中的（i）），但仍然会遭受可能超过 100ms[16]的无线延迟（图 4.1中的（ii）和（iii））。此外，主动队列管理算法大多是为了**丢弃**一些数据包，而许多现代的拥塞控制算法算法则是为了对增加的数据包**延迟**做出反应，而对数据包丢弃不敏感[49,57,60]。这也可以在图 4.3(a)中验证：CoDel 几乎无法改善延迟敏感的拥塞控制算法（如 Copa）的性能。还有一些解决方案试图将主机和网内路由器一起进行设计，以从网络中获得更好的反馈，包括 XCP[75], RCP[76]、Kickass[77]和 ABC[78]。然而，它们的设计目标是从路由器中获得网络条件的精确估计，而收集的信息仍然需要经过完整的控制回路。4.7节还将 Zhuge 的性能与 ABC 进行了比较，以展示主机-路由器协同设计的潜在改进空间。

4.2.3　工作思路：减少控制回路

本章的关键见解是尽早感知网络条件，及时将条件反馈回发送端，以**最小化控制回路**，并以可部署的方式执行上述操作。这里有如下三点观察：

（1）一个数据包在到达时就知道它的命运。在大多数情况下，当一个数据包到达瓶颈队列时，它可以通过观察整个队列来预测它的延迟。例如，数据包的排队延迟可以通过将队列长度除以出队速率来大致估计。因此，当出队速率减少时，我们可以猜到后续数据包到达时的排队延迟增加。与其他后续信号（如数据包丢失或测量的排队延迟）相比，估计的排队延迟是最早的可用带宽降低的信号。因此，本章希望利用这个最早的信号及时控制发送速率并适应可用带宽下降。

（2）快速将最早的信号传回发送端。仅仅找到可用带宽下降的信号是不够的。我们需要快速将这个信号传回发送端。理想的解决方案是直接**告诉**发送端瓶颈队列的当前状态。这样，这个信号就可以绕过控制回路的膨胀部分（图 4.1中的（i），（ii）和（iii））。同时，无线接入点的上行队列（图 4.1中的（iv））和广域网的延迟（图 4.1中的（v））的延迟通常是稳定的。无线接入网的上行通常是连接到互联网的有线线缆，通常具有数百 Mbps 的容量。广域网延迟（图 4.1中的（v））则是最后一跳接入点和发送端之间的延迟。但注意到，有线网用户也会有这两部分控制回路。如我们在图 1.5中对有线网测量的结果所示，这些控制回路相对稳定。

（3）只修改最后一跳接入点是可部署的。回顾传输层设计的历史，有一系列出色的努力，但不幸的是由于实际问题并没有广泛部署。例如，XCP[75]、RCP[76]、Kickass[77]、ABC[78]和主动网络[95]在过去二十年中都需要对服务器和一些或所有路由器进行修改。然而，服务器通常由内容提供商（例如，Google、Facebook）控制，而路由器则由设备制造商（例如，Netgear 生产无线接入点）控制。协调所有这些各方来推动新的传输创新是即使是可能也是非常具有挑战性的。与上述工作

不同，Zhuge 仅修改最后一跳的接入点，这可以减少部署的障碍。无线接入点制造商可以单独部署并观察性能的改进，而无需与内容提供商协调。此外，从家庭用户的角度来看，最后一跳的接入点是他们可以控制的唯一地方。因此，我们的设计可以在家庭用户的设备上实现，而无需与运营商协调。

4.3　早反馈机制设计

本节介绍通过 Zhuge 来控制无线延迟抖动的设计挑战和框架概述。

4.3.1　设计挑战

Zhuge 通过减少控制回路来处理无线延迟抖动。然而，Zhuge 有如下两个设计挑战。

（1）多媒体流量及时、准确的逐包延迟估计。 Zhuge 通过在无线最后一跳到达数据包到达的时候预估未来的延迟来尽早获取网络状况。为了正确地指导发送方的拥塞控制算法进行发送速率自适应，需要对每个包进行准确的延迟估计。然而，对于无线环境中的实时流量，准确的延迟估计是具有挑战性的，因为由于两个原因，瓶颈队列在**亚 RTT 粒度**上处于瞬时波动状态。

- **实时流量的包到达是突发的。**实时应用程序以视频帧为单位生成内容。为了减少端到端的延迟，发送方倾向于突发地发送同一帧的包。这表明即使在稳态下，队列也可能会很快地建立起来。
- **无线信道的包离开是突发的。**无线网络的共享性导致无线信道的争用，这会导致无线信道的传输速率不稳定。因此，即使在稳态下，无线信道的传输速率也会波动。

一个简单的估计方法是将队列长度除以出队速率。然而，这种方法面临着**瞬态-稳态权衡**[99]：出队速率通常是在滑动窗口（例如 WiFi 中的 40ms[78]）上进行测量的，但这个窗口的设置就面临着权衡。短的窗口会导致稳态下的测量变化，而很长的窗口则会丢失瞬态延迟波动在亚 RTT 这样一个时间粒度上的特性。因此，对于实时流量，很难在无线最后一跳准确地估计每个包的延迟。

（2）有效的消息反馈给各种协议和拥塞控制算法。 Zhuge 的目标是尽快地将网络状况告知发送方。一个直接的解决方案是构造一种新的反馈包发送给发送方。然而，对于大多数在实际部署的拥塞控制算法，网络状况（例如当前可用带宽）并不会在互联网上显式地传递。直接将网络状况告知发送方需要同时也修改发送方，以使发送端的拥塞控制算法能够理解这一消息。正如上文所述，我们希望能够在

不修改发送方的情况下实现基于接入点的解决方案，以实现规模化部署。

更具有挑战性的是，实际部署的拥塞控制算法和协议的多样性。传输协议的包头可能是未加密的（例如 TCP）或加密的（QUIC）。为了实现更低的延迟，实时多媒体应用程序倾向于自定义拥塞控制算法，这些算法依赖不同的信号来调整发送速率。例如，一些应用程序会在内核中修改 TCP 拥塞控制算法[100]。对于基于 WebRTC 的应用程序，网络状况会定期汇总到特殊的反馈包中[25]。各种拥塞控制算法使得有效地将网络状况传递给发送方变得具有挑战性。

4.3.2　框架概述

为解决上述挑战，我们在 Zhuge 中设计了两个基本模块：**命运预测器**和 **反馈更新器**。

为了实现及时、准确的逐包延迟估计，本章在第4.4节中引入 Zhuge 命运预测器来在每个数据包到达的时候预测每个包的未来延迟。为了打破前面所说的稳态-瞬态的权衡并获得逐包延迟估计，我们将延迟分解为不同的部分，并引入了长期和短期两个估计器。我们测量平均出队速率来计算长期排队延迟，并测量队列前端的包的停留时间来响应短期波动。

为能够及时地将预测的网络状况告知发送方，本章在第4.5节中引入 Zhuge 反馈更新器来将预测的网络状况转换为发送方可以理解的信号。本章将现有的协议分类为带外反馈和带内反馈。对于带外反馈协议，到达反馈包是信号（例如 TCP 中的 ACK 包）。对于带内反馈协议，反馈包的有效载荷中携带了网络状况，例如 WebRTC 中的 Transport-wide Congestion Control Feedback（TWCC-FB）包[101]。因此，Zhuge 为不同的协议设计了不同的反馈机制来将延迟传递给发送方。

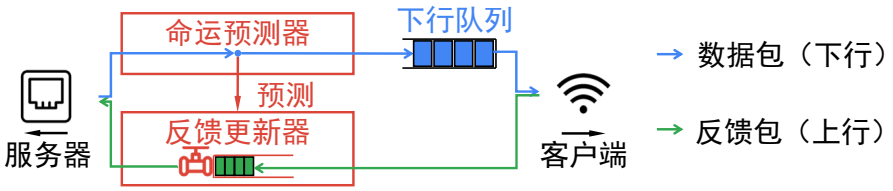


图 4.4　在最后一跳无线接入点中 Zhuge 的整体工作流程

Zhuge 的整体工作流程如图4.4所示，其中的命运预测器和反馈更新器是 Zhuge 的主要模块。当一个数据包到达无线接入点的以太网端口时，命运预测器会预测其未来的延迟，并将数据包转发到下行队列。反馈更新器会将预测的网络状况更新到反馈包中。如果一个新到达的数据包观察到网络状况下降（例如队列长度增加），则可以立即将估计的无线延迟去施加于反馈包中，从而实现逐包延迟估计。这样，最早的信号就可以绕过控制回路（图 4.1中的 (i)-(iii) 部分）的排队延

迟和无线传输延迟，传递给发送方。

4.4　命运预测器

预测一个数据包的命运，即预测其到达客户端的延迟。在无线网络中，这个延迟可以分解为两个部分[102]：(i) 排队延迟：从数据包到达接入点到数据包离开队列到底层驱动程序（即网络层中的延迟）。(ii) 传输延迟：从数据包被传递给无线驱动程序到数据包到达接收方（即链路层中的延迟）。接下来我们介绍如何及时预测这两个延迟。

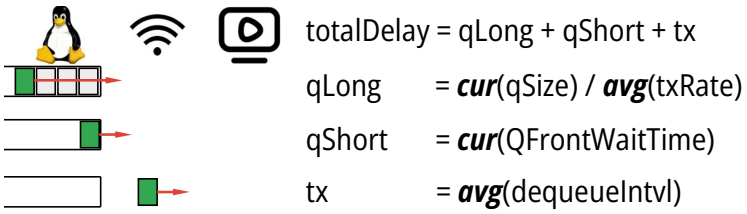


图 4.5　命运预测器估计延迟时的不同的延迟组件

4.4.1　排队延迟预测

如第4.3.1节中讨论的，将队列长度除以出队速率的简单方案会面临瞬态-稳态的权衡。短的滑动窗口会导致由于到达和离开的突发导致预测延迟的剧烈波动，而长的窗口则无法快速检测网络状况的变化。为了解决这个问题，我们分析了如何捕获这两个原因导致的延迟波动。

- **实时多媒体流量的突发到达。**实时多媒体流量可能会快速堆积无线队列，增加其长度。我们的设计选择是预测每个数据包的命运，而不是基于周期地在宏观尺度上预测。
- **无线信道的突发离开。**突发的数据包离开会在毫秒级别上引入传输速率的瞬态波动，这些波动很容易被滑动窗口平均掉，因此在现有的滑动窗口测量中很容易被忽略。

因为这两个原因导致的延迟波动是不同的，我们将排队延迟分解为两个部分：长期排队延迟（`qLong`）和短期排队延迟（`qShort`），如图 4.5所示。具体来说，`qLong`是从数据包到达接入点到数据包到达队列前端的时间，用于覆盖由于无线竞争和实时多媒体流量的突发导致的延迟波动。在这里可以通过当前队列长度除以平均出队速率来估计 `qLong`，因为它更受队列动态的影响。短期排队延迟是从数据包到达队列前端到数据包最终出队的时间。`qShort` 更多地与链路层的发送模式有关（例如，MAC 层数据单元的聚合会导致波动）。因此我们可以通过当前队列长度除以平均入队速率来估计 `qShort`，然后将它们相加得到预测的排队延迟。在

图 4.5中，$avg(\cdots)$ 表示滑动窗口内的平均值，$cur(\cdots)$ 表示在计算时刻的值。$qSize$ 是队列长度，$qFrontWaitTime$ 是当前队列首部已经等待的时间，$txRate$ 是队列的出队速率。

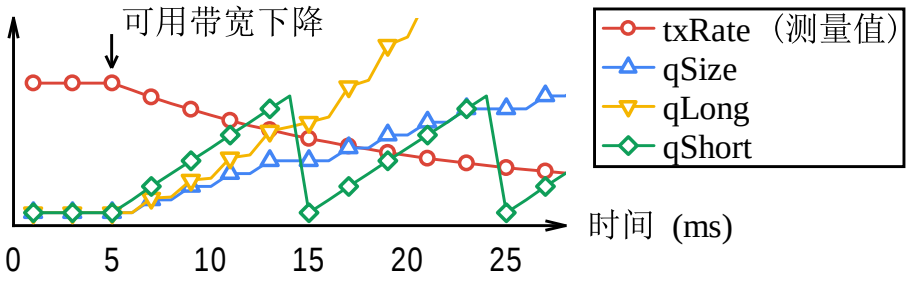


图 4.6　qLong 和 qShort 在 5ms 时刻的可用带宽下降的反应

使用长期和短期排队延迟的组合有两个优点。在这里用一个例子来说明这两个优点，如图4.6所示。首先，使用 qShort 可以快速检测到可用带宽的下降。当可用带宽开始下降时，由于队列需要一些时间来建立，而测量的传输速率也需要一些时间来下降，因为滑动窗口的原因，qLong 的增长会很慢。相反，数据包需要等待更长的时间才能发送，这可以立即被观察到。如图 4.6中的 5-15ms 的区间所示，qShort 将主导总排队延迟的增加，从而快速反映了可用带宽的下降。第二，使用 qLong 可以提供对队列已经建立的稳定和准确的排队延迟估计。例如，当队列仍然过载时（例如，图 4.6中的 15ms 之后），qLong 会主导排队延迟，提供一个稳定、精确的估计。

接下来，我们进一步介绍如何处理两个实现排队延迟估计的实际问题。

第一是面向突发离开的调整。队列的突发离开可能会影响 qLong 的估计准确性：当队列中有多个数据包时，它们可能会一起发送。事实上，根据我们的设计，突发内的波动应该反映在 qShort 上。因此，当计算 qLong 时，我们估计 $qSize$ 为：

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

其中，$sizeOfPacketsInQueue$ 是队列中的数据包数量，$maxBurstSize$ 是 1ms 内最大的离开数据包数量。

第二是队列调度的计算。在实践中，如研究论文[78]中所假设的那样，队列可能不是按照先进先出的方式排列的。例如，systemd 中的默认队列调度在不同的 5 元组流之间并不见得会保序[103]。对于蜂窝网络，每个流也有自己的队列，与竞争流隔离[78]。在这种情况下，我们需要计算实时音视频流对应队列的统计信息。

4.4.2　传输延迟预测

本文主要关注 WiFi 网络中的延迟估计。本文推荐读者参考 [78] 中的关于蜂窝网络的延迟估计。预测每个数据包的传输延迟是具有挑战性的，因为它与底层无线驱动程序和物理通道相关。特别是对于高性能的无线设备（例如，802.11ax 协议的设备），关键特性（例如，比特率选择和帧聚合）被编码在硬件设备中，并且在不与供应商进行大量交互的情况下，无法从接入点 CPU 访问[16]。例如，许多 Netgear 路由器采用 Qualcomm Atheros 硬件[104]，其中一些与性能相关的关键特性（帧聚合等）是直接被硬编码到固件中的，一般用户无法访问。因此，预测无线通道的传输延迟是具有挑战性的。

根据 [78]，我们总结了传输延迟的以下观察结果。首先，与所有链路层协议类似，无线通道中应该只有**一个数据单元**。例如，802.11ac 发送端可能会将多个数据包聚合成一个数据单元（聚合 MPDU，或称之为 AMPDU）。但是，多个 AMPDU 不能同时传输，因为它们的信号会相互干扰。因此，无线驱动程序将多个数据包聚合成一个 AMPDU，将其发送出去，并等待该 AMPDU 的确认或超时。其次，随着近期在 Linux 主线中的努力，无线网络堆栈的下层队列已经暴露给队列调度[102]。在这种情况下，无线网络堆栈中的下层队列仅用于将多个数据包聚合成链路层帧。

因此，如图 4.5所示，传输延迟 `tx_delay` 通过网络层队列中数据包出队的平均时间间隔计算，该时间间隔与 txRate 相似。滑动窗口应该足够长，以覆盖发送方的至少两个突发，以便持续测量数据包。请注意，由于在一瞬间可能会同时聚合多个数据包一同出队，因此我们不会计算小于一毫秒的时间间隔。

4.5　反馈更新器

Zhuge 将估计的延迟以一种发送端可以理解的方式返回给发送端。为了避免在端系统上进行修改，Zhuge 遵循应用协议和拥塞控制算法的原始反馈消息格式。本节首先对流媒体应用程序的常见拥塞控制算法的反馈机制进行分类（§4.5.1），然后介绍我们相应的解决方案（§4.5.2 和 §4.5.3）。

4.5.1　反馈机制分类

我们调查了一些实时流媒体应用程序，总结了它们的反馈机制（见表 4.1）。它们可以分为两类，带外和带内反馈。我们在图 4.7中展示了它们的行为。① 带外反馈协议不显式地将反馈信息搭载在数据包负载中，但带内反馈会。蓝色和白色方

① 一些协议可能同时使用两种反馈机制。例如，RTP 发送端还会类似于 TCP 那样自己测量 RTT[25]。这个 RTT 信息不用于速率控制，而是用于在 RTP 中稳定控制回路。

(a) 带外反馈　　(b) 带内反馈

图 4.7　带内反馈与带外反馈的示意图

表 4.1　现有实时多媒体应用的反馈机制的分类

	协议	拥塞控制算法	应用
带外反馈 (§4.5.2)	TCP, QUIC[105]	PCC[50] BBR[57] Copa[49]	Meta Live[59] Windows 365[106] Twitch[107] Tencent Start[100]
带内反馈 (§4.5.3)	RTP+RTCP[25]	GCC[60] NADA[47] Scream[48]	Google Stadia[108] Zoom[42] Microsoft Teams[109]

块代表数据包头和负载。

- **带内反馈**。如图 4.7(b)所示，带内反馈意味着反馈信息显式地写在特定类型反馈包的负载中。例如，RTP 和 RTCP 协议就是带内反馈。接收端记录每个数据包的到达时间，并定期构造反馈包将时间间隔传回发送端[101]。
- **带外反馈**。带外反馈机制不显式地将速率控制相关的信息写在反馈包的负载中。相反，发送端在收到反馈包后自己计算所有网络状况。例如，TCP 客户端会确认收到的每个数据包。当发送端收到 ACK 包时，它会计算 RTT、接收速率和其他网络状况。

我们分别为上述两种反馈机制设计解决方案。对于带外反馈机制，网络状况只在发送端测量。我们的观察是，我们可以**故意延迟**反馈 ACK 包，将网络状况带回来。对于带内反馈机制，因为反馈信息写在反馈包的负载中，我们需要更新反馈包的负载。接下来我们详细介绍两种解决方案。

4.5.2　带外反馈：延迟 ACK 包

ACK 包用在依赖于带外反馈的应用中，但是不同的拥塞控制算法对 ACK 包的处理方式不同。例如，BBR 会统计接收速率并查询 ACK 包中的最小 RTT 用于速率调整，而 Copa[49] 对每个数据包的延迟敏感。为了满足不同拥塞控制算法的需求，我们的设计目标是**延迟 ACK 包**而非改变它们，以便在最细粒度的每个数据包中传递估计的延迟。拥塞控制算法可以聚合细粒度的信息并以自己的方式做出反应。

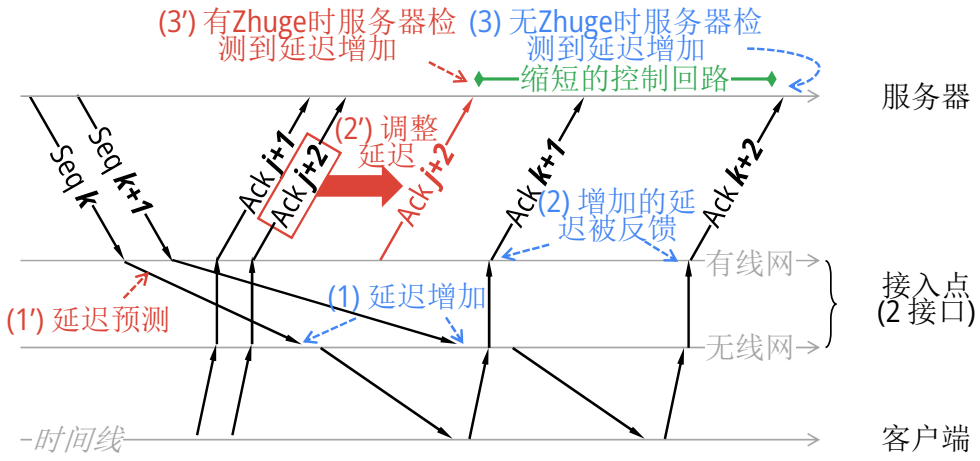


图 4.8　通过在反方向上延迟反馈包来携带预测的命运的示意图

图 4.8展示了一个在无线接入点上，Zhuge 如何将预测的数据包的信息传递回去视图的例子。Zhuge 在反方向上立刻将反馈包延迟，以将预测到的数据包命运携带回去。蓝色箭头表示发送端如何不使用 Zhuge 来感知网络状况。假设序号为 k 和 $k+1$ 的数据包从服务器到达无线接入点，现在可用带宽下降。如果不使用 Zhuge，那么后面的数据包（序号 $k+1$）将会比预期的更晚出队，排队延迟将逐渐增加（蓝色的 (1)）。客户端将会收到这两个数据包的间隔变大，因此会用这个间隔来确认它们。确认包将会以这个间隔到达 AP 并离开（蓝色的 (2)）。如图 4.9 所示，不使用 Zhuge 时，发送端只能在延迟数据包的确认包到达时（*deltaDelay*）才能确认增加的 RTT。

有 Zhuge 时，我们可以预测序号为 k 和 $k+1$ 的数据包的延迟，因此可以在到达无线接入点时预测它们的延迟（红色的 (1')）。如果预测延迟增加，我们可以立即延迟之前到达或将到达接入点的确认包。如图4.8中的红色箭头所示，我们可以故意放大其他确认包之间的间隔（确认包 $j+1$ 和 $j+2$），以便及时通知发送端（红色中的 (2')）。这样，服务器就可以在数据包到达服务器时检测到可用带宽下降（红色的 (3')）。服务器测量的不同数据包的 RTT 曲线将向前移动，如图4.9所示。在图4.9中，Zhuge 通过延迟更早返回的 ACK 包来快速反馈网络条件。*actualDelay* 是 Zhuge 的控制回路。因此，拥塞控制算法的控制回路将减少 $(k+1)-(j+1)$（以确认包的数量计数，图 4.8中的绿色箭头）。还要注意，Zhuge 不需要查看和匹配序列号和确认号——这里展示的数字仅用于说明概念。相反地，Zhuge 只查看五元组来识别流，并把序列号和 ACK 号作为黑盒子来处理。这样，即使传输协议是加密的（如 QUIC），Zhuge 依然可以工作。

然而，下行数据包和上行反馈包是异步到达接入点的。因此，很难将下行数据包预测的延迟一对一映射到上行反馈包。这可能会影响网络条件的精度。当数据包

到达时，命运预测器将根据当前网络条件进行更新。更新的队列条件包括 qLong，qShort 和 tx，如 §4.4 中介绍的那样。最终预测的总延迟计算如下：

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

下面，我们介绍 Zhuge 的设计原则，以确保数据包的延迟精度。

（1）在稳态中传递精确的长期延迟。因为 Zhuge 故意延迟了上行反馈包，因此一个自然的担心是这种延迟是否会影响网络 RTT 的稳态估计。例如，图 4.8中的数据包序号 $k+1$ 已经在下行方向中受到了长时间的排队延迟。如果 Zhuge 在上行方向中也引入了非常大的延迟，那么它将会夸大真实的 RTT，并可能干扰拥塞控制算法的估计。

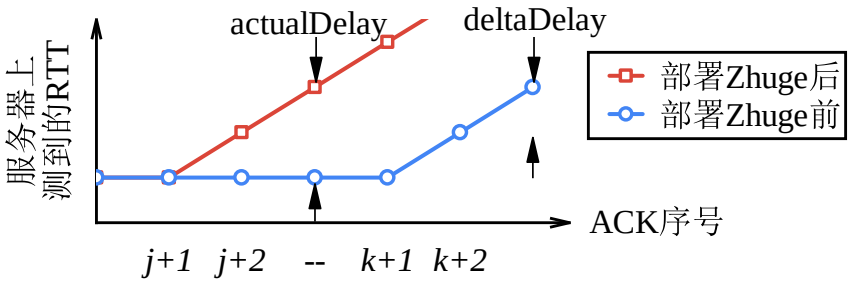


图 4.9　Zhuge 将逐包 RTT 曲线向左平移的示意图

为解决这一问题，我们不直接将下行方向中的延迟估计绝对值添加到上行方向中的额外 ACK 延迟中。相反，我们记录**相对延迟差**，即下行数据包之间的延迟差。当估计的延迟在增加时，我们可以从下行方向记录一系列正的延迟差，并逐渐增加上行方向中的延迟。当队列已经稳定建立（例如，对于序列号 $k+1$ 之后的数据包），延迟差将大约为零，上行方向中的反馈数据包也不会受到额外的延迟。

（2）传递精确的短期延迟波动。短期逐包延迟动态对于像 Copa 这样的延迟敏感的拥塞控制算法非常重要。这些拥塞控制算法将利用 RTT 以下的延迟模式来控制发送速率。但是，直接利用延迟差机制可能无法准确传递短期延迟波动。原因是短期延迟是逐包变化的。并非每个延迟差都可以在单独的 ACK 中传递。这可能会导致多个延迟差累积到一个 ACK 中，从而导致不可靠的延迟估计结果。例如，当三个数据包在接入点中以每个包之间的延迟差为 +1ms 到达时，直接延迟下一个 ACK 3ms 将会引入比实际值更大的延迟增加。

为了解决这一问题，我们不直接传递逐包延迟差，而是追求下行延迟差和上行 ACK 延迟之间的 **分布等价性**。我们维护最近下行数据包的延迟差分布。当下行数据包到达时，我们根据命运预测器的预测延迟计算延迟差。当上行反馈数据包到达接入点时，我们从最近的延迟差分布中**采样**，并使用采样得到的值来延迟反

算法 4.1　数据包到达时：带外反馈

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

算法 4.2　ACK 包到达时：带外反馈

```
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 计划在当前 ACK 被发出 actualDelay 后再发送
12 lastSendTime = curArrvTime + actualDelay
```

馈数据包。这样，即使在数据包到达和离开时存在突发性，Zhuge 也能模仿反馈数据包的延迟分布。

（3）保持反馈包的顺序。 Zhuge 的延迟差机制会引入一个新的问题：如何保持反馈包的顺序。例如，如果 ACK $j+1$ 和 $j+2$ 同时到达，且 ACK $j+2$ 的延迟比 ACK $j+1$ 小，AP 可能会先发送 ACK $j+2$，这会导致反馈包的顺序混乱。如果将后续反馈包的发送时间延迟到与前面的包相同，例如等待 ACK $j+1$ 发送后再发送 ACK $j+2$，这会导致 RTT 的过度估计。

为了解决这一问题，我们引入了一个延迟令牌来保持反馈包的顺序，同时避免 RTT 的过度估计。当需要让后续反馈包等待前面的包发送时，我们将等待时间存储为一个延迟令牌。下次采样到正的延迟差时，我们会先尝试消耗令牌。这样，实际延迟的平均值将与预测延迟的平均值保持一致。

我们最终展示了 Zhuge 反馈更新器如何使用预测的延迟来更新反馈数据包。如算法4.1 所示，当每个数据包到达时，给定该数据包的预测延迟，Zhuge 首先计算延迟差（第 1 行）。如果延迟差为非负数，我们将其存储到一个滑动窗口中。由于 Zhuge 只能延迟正的时间，如果延迟差为负数，我们需要将其存储为令牌

（第 4-5 行）。异步地，当每个 ACK 包到达时，将执行算法4.2来适当地延迟 ACK。curArrvTime 是当前 ACK 的到达时间戳，lastSentTime 是从无线接入点发送到服务器的最后一个 ACK 的计算发送时间戳。为了保持顺序，Zhuge 首先计算当前 ACK 包的最小延迟，以确保当前 ACK 包在前面的 ACK 包之后发送（第 1 行）。Zhuge 然后从最近的滑动窗口中随机采样一个延迟差（第 2 行）。Zhuge 进一步检查是否有未消耗的令牌，并在有的情况下消耗令牌（第 3-10 行）。最后，当前 ACK 包将被延迟并在 actualDelay 之后发送。

4.5.3　带内反馈：更新负载

对于如 RTCP[25] 这样的带内反馈机制，反馈信息（例如每个数据包的接收时间）被写入反馈数据包的负载中。我们需要更新它们的负载以将新估计的延迟带回发送者。我们使用 RTP（数据）/RTCP（反馈）协议对来介绍如何使用两个步骤更新反馈数据包。

- **第 1 步：数据包预测。**每当 RTP 数据包到达时，Zhuge 将预测其延迟并将预测的延迟与其 RTP 传输范围拥塞控制（TWCC）序列号一起存储在 RTP 头中。
- **第 2 步：反馈构造。**当需要将当前网络状况反馈回发送者时（例如每个 RTT 或每帧[101]），Zhuge 将像 RTP 接收器一样行为，并基于存储的延迟和序列号构造 TWCC 反馈数据包。为了确保时间戳一致性，Zhuge 将仅发送自己构造的 TWCC 数据包，并丢弃来自客户端的所有 TWCC。对于其他类型的反馈数据包（例如用于损失恢复的负面确认（Negative Acknowledgment，NACK），接收者报告（Receiver Reports，RR）等），Zhuge 将像往常一样将其从客户端转发到服务器。

具体的 RTP/RTCP 数据包格式在 RFC[25,101] 中进行了介绍。同时，Zhuge 带内反馈机制的实现解决了两个实际存在的问题。

（1）时间同步。由于接入点上的时间戳可能与接收器不同步，因此一个直接的问题是无线接入点和接收端之间的时间差是否会影响对拥塞控制算法的估计。事实上，服务器被设计为容忍服务器和反馈数据包构造者（无论是客户端还是接入点）之间的时间差，因为服务器也不与客户端同步。因此，Zhuge 只需丢弃来自客户端的所有 TWCC 数据包即可，因为生成的 TWCC 数据包的时间戳来自同一个无线接入点上的时钟，这个与服务器是一致的。

（2）端到端加密。在某些情况下，RTP 数据包和 RTCP 反馈数据包可能会被端到端加密[110]。Zhuge 可以在这种情况下工作，原因如下。首先，Zhuge 不需要解

密 RTP 数据包负载。相反，Zhuge 只需要记录序列号，这些序列号在包头中是明文存储的。其次，Zhuge 也不需要解密 RTCP 反馈数据包负载。Zhuge 只需要**加密**构造的反馈数据包，以便服务器可以正确解码该数据包。幸运的是，在实践中，许多服务器和客户端在连接开始时以明文共享公钥[110]。Zhuge 可以拦截并保存服务器的公钥，并使用它来加密构造的反馈。

4.6　讨论

在这里，我们讨论 Zhuge 在实际部署中的考量与一些局限性。

首先是瓶颈到底在最后一跳还是第一跳的问题。我们主要介绍和评估 Zhuge 在下行方向的性能，即无线网络作为**最后一跳**。这是因为对于远程桌面、云游戏和点播等许多实时多媒体传输应用程序，视频是从服务器传输到客户端的。远程服务器作为发送者可以调整发送速率，并且受到长控制循环的影响。对于其他点对点的实时多媒体传输应用程序，如视频会议，无线网络作为**第一跳**也可能会引入尾延迟。在这种情况下，客户端中会建立队列。Zhuge 中的机制也可以用来处理第一跳尾延迟，方法是通过操纵客户端网络堆栈来实现，这需要与应用程序进行集成，超出了我们的范围。

其次是可能引入的公平性的问题。减少拥塞控制算法的控制循环意味着对网络条件的反应更快，这可能意味着发送速率增加和减少可能会更加激进。一个自然的担忧是 Zhuge 是否会损害优化流与其他流之间的公平性。我们的答案是**否**，因为 Zhuge 不会优先考虑目标流而牺牲其他流。(1) 当**发送速率增加**时，无线队列应该接近空闲。在这种情况下，Zhuge 优化的流与没有 Zhuge 的流具有类似的控制回路，不会变得更具侵略性。(2) **发送速率减少**可能是由于无线队列积压而导致的。Zhuge 只是减少了控制回路并加速了收敛，而不同拥塞控制算法之间收敛的公平性应该在拥塞控制算法的设计中处理[111]。我们在 §4.7.6 中进一步评估了 Zhuge 的公平性。

最后是可扩展性的问题。在本文中，我们提出了一种能够广泛应用于 TCP 和 RTP/RTCP 协议的广泛应用的解决方案。但是，未来可能会出现新的协议。对于新的带外协议，只要我们能够从数据包中识别流信息，Zhuge 就可以从网络层工作。例如，由于我们不需要知道数据包的特定序列号，即使 QUIC 对所有数据包进行端到端加密，Zhuge 仍然可以与 QUIC 一起工作。对于带内协议，Zhuge 需要运营商发布这种协议的格式，以便相应地修反馈更新器的逻辑。

4.7　实验评估

我们首先在4.7.1节中介绍了 Zhuge 的实现细节，以及实验设置，然后通过以下问题来评估 Zhuge 的性能：

- **在真实无线数据集中，Zhuge 是否可以改善尾延迟？**我们使用五个真实跟踪数据集评估了 Zhuge 的性能。评估结果表明，Zhuge 可以将长尾延迟的比例降低 75%，并提高应用程序性能 91%。(§4.7.3)
- **在不同类型的无线竞争下，Zhuge 的性能如何变化？**我们设计了带宽减少、流竞争和无线干扰的无线场景。我们观察到 Zhuge 在所有场景下都能提高性能。(§4.7.4)
- **在真实世界中， Zhuge 可以带来多大的性能提升？**我们在办公室环境中部署了 Zhuge 的原型，发现 Zhuge 可以从 17% 到 94% 提高网络和应用指标。(§4.7.5)
- **Zhuge 在稳态性能、公平性和 CPU 资源上的成本如何？**我们发现 Zhuge 并不会削弱实时多媒体流的稳态性能和公平性，而且 CPU 资源成本都很低。(§4.7.6)

4.7.1　实现细节

我们将 Zhuge 实现在一个基于 NS-3 的仿真器和一个真实无线接入点的测试平台上。在仿真器中，我们根据 WebRTC 中的参考实现实现了简化的视频编码器和解码器。我们实现了 RTP/RTCP 和 TCP 协议栈，以及 §4.7.2 中列出的高级拥塞控制算法和主动队列管理算法。我们构建了网络层和链路层无线队列，并为仿真器实现了 Zhuge。我们将命运预测器和反馈更新器中的滑动窗口设置为 40ms，这是因为我们的视频流是 25fps。我们将发布我们的仿真器和实现的算法。对于测试平台实验，我们在 OpenWrt 中实现了 Zhuge，OpenWrt 是嵌入式网络设备的开源操作系统。命运预测器和反馈更新器以用户空间特性的形式实现在 OpenWrt 中，它们使用数据包套接字来观察和修改数据包。我们通过匹配 IP 地址来识别目标实时多媒体流，该 IP 地址是由 Zhuge 维护的可配置 IP 列表[112-113]。我们使用 Netgear WNDR 3800 路由器[104]，该路由器运行 OpenWrt 并支持 WiFi 802.11n，用于性能评估。我们还在 TP-Link 路由器上部署 Zhuge 并测量了 CPU 资源。

4.7.2　实验设置

我们生成了 1080p 24fps 的视频，平均比特率为 2Mbps。我们在下面列出了我们使用的基线、跟踪和指标。

基线。为了充分评估 Zhuge 的性能，我们实现了 RTP/RTCP（带宽反馈）和 TCP（不带宽反馈）协议栈。Zhuge 可以与高级拥塞控制算法和主动队列管理（AQM）机制一起使用。在我们的 RTP/RTCP 评估中，我们实现了以下解决方案：

- *Gcc+FIFO.* Google Congestion Control (Gcc)[60] 是 WebRTC 的默认拥塞控制算法，也被许多应用程序（如 Google Stadia 和 Google Meet）采用。因此，我们将 Gcc 作为 RTP/RTCP 协议的拥塞控制算法，并使用无线队列中的 FIFO 调度器作为基线。
- *Gcc+CoDel.* CoDel[68] 是一种主动队列管理机制，旨在处理队列溢出。它会丢弃队列前面的数据包，而不是队列尾部的数据包，当排队延迟超过目标值时。
- *GCC+Zhuge(+CoDel).* 我们在 RTP / RTCP 上实现了 Zhuge，并评估当其与 Gcc 共同工作时的性能。

对于 TCP 的评估，我们实现了以下解决方案。请注意，我们选择的拥塞控制算法是无损的。因此，为了简洁起见，我们分别在 FIFO 和 CoDel 上评估每个解决方案，并选择更好的执行者作为基线。

- *Copa.* Copa[49] 是一种针对 TCP 的延迟敏感拥塞控制算法。根据许多实验，它可以实现低延迟[78,93]，并已部署在实际的流媒体服务中[59]。
- *Copa+FastAck.* FastAck[16] 是一种基于 WiFi 无线接入点的优化，它通过伪造 TCP ACK 包来减少延迟，该包在从客户端设备接收到 802.11 ACK 时发送。
- *ABC.* ABC[78] 通过网络主机协调来优化无线网络性能。它直接从访问点检测网络状况，并将其报告给发送方。但是，ABC 需要修改无线接入点。
- *Copa+Zhuge.* 我们在 TCP 上实现了 Zhuge，并评估其与 Copa 的性能。

数据集。我们使用了五个亚秒级精度的真实世界的链路情况数据集。其中两个来自 WiFi 网络，三个来自蜂窝网络。跟踪数据集记录了每个时间戳的带宽和延迟。

- W1 - 餐厅 WiFi。我们测量了一个拥挤的餐厅提供的公共 WiFi AP 的吞吐量[114]，并在 3 小时的晚餐期间计算 200ms 分辨率的吞吐量。WiFi AP 在 2.4GHz 上运行 802.11ac。
- W2 - 办公室 WiFi。我们还测量了办公室中 WiFi 无线接入点的吞吐量，该 AP 在办公室工作时间运行 10 小时。我们的办公室无线接入点工作在 5GHz 频段，使用 802.11ac（WiFi 5）。
- C1 - 室内混合 4G/5G。通过 4G 和 5G 蜂窝网络的室内场景测量吞吐量[96]。
- C2 - 城市 4G 和 C3 - 城市 5G。文献[97] 收集了 4G 和 5G 蜂窝网络的数据

包。我们根据标签将数据集分为 4G 和 5G。

指标。我们在评估中使用以下指标。

- **RTT**。我们在网络层测量数据包的往返时延（RTT）。我们将 RTT 大于 200ms 的部分来统计尾部延迟比率。
- **帧延迟**。帧延迟是指发送端编码帧和接收端解码帧之间的时间间隔。一帧只能在所有该帧的数据包都到达并之前的帧已经解码之后才能解码。因此，帧延迟是评估视频的延迟相关用户体验的直接指标。我们将帧的端到端延迟大于 400ms 的帧视为延迟帧。
- **帧率**。如果客户端到达的帧率太低，用户也会体验卡顿。因此，我们还可以根据帧率来评估视频质量。我们将每秒帧率 <10fps 视为低帧率。

本文中，我们并未采用视频质量指标，如 PSNR[31]、SSIM[30] 和 VMAF[115]，因为它们并不反映端到端的交互延迟。最近的一些工作在尝试去设计把延迟考虑在内的主观体验指标[65]，本章对这些暂时不考虑。

4.7.3　基于真实数据集的仿真

我们用 NS-3 模拟器评估了 Zhuge 在真实世界无线带宽数据集上的尾部网络延迟和应用性能。我们使用五个跟踪数据集，分别模拟了瓶颈链路。我们评估了 Zhuge 在 RTP/RTCP 和 TCP 上的性能。

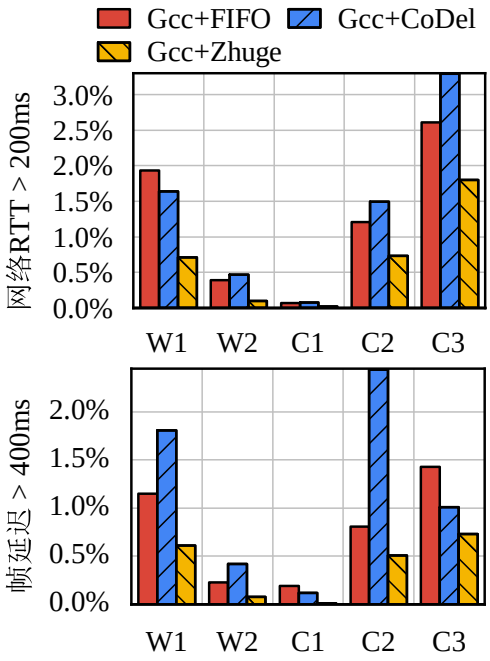


图 4.10　在 RTP/RTCP 协议上基于真实网络数据集的仿真结果

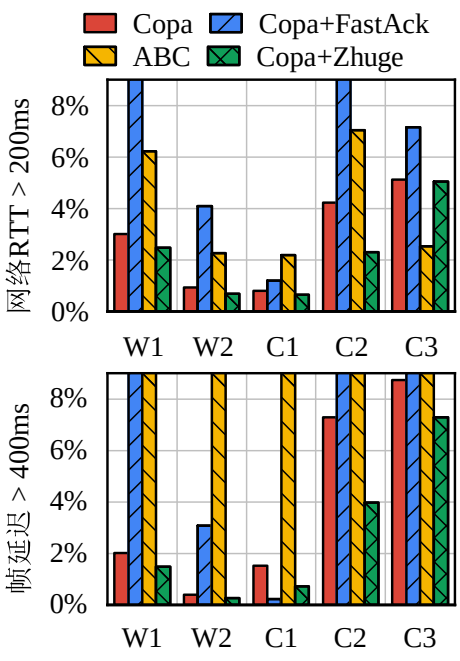
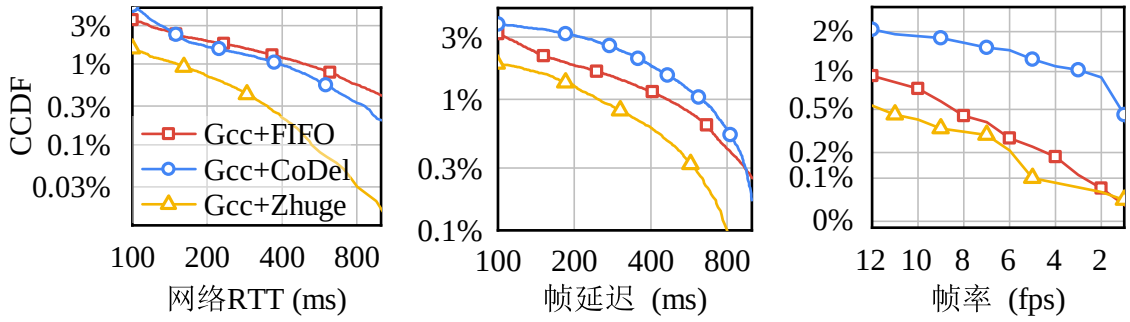
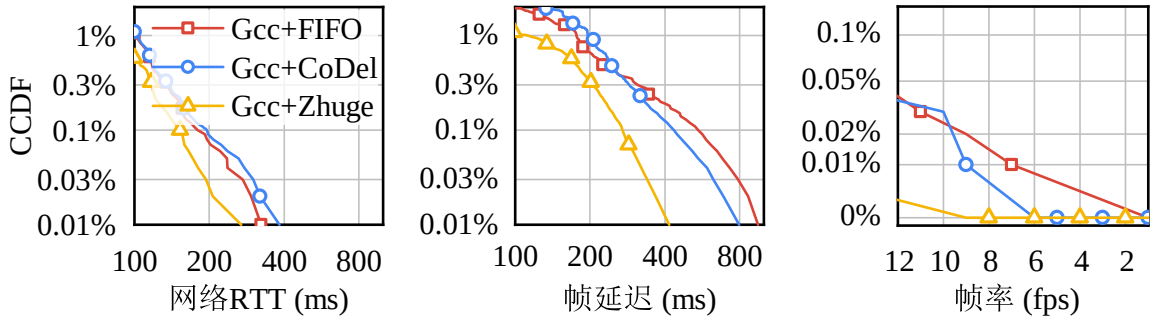


图 4.11　在 TCP 协议上基于真实网络数据集的仿真结果

首先是 RTP/RTCP 协议。如图4.10所示，对于 RTP/RTCP，Zhuge 在所有的数



(a) 数据集 W1 – 餐厅 WiFi



(b) 数据集 C1 – 室内 4G/5G 混合

图 4.12　Zhuge 和其他基线在 RTP/RTCP 协议上的延迟分布

据集中都优于所有的基线算法，且具有一致的低延迟。具体来说，Zhuge 可以将网络 RTT 大于 200ms 的比例降低 45% 到 75%。因此，不同数据集中延迟的帧比例降低了 38% 到 92%。这显著减少了视频的卡顿，并提高了用户体验。我们还观察到，Gcc+CoDel 在数据集 C1 和 C3 中优于 Gcc+FIFO，但在其他三个数据集中表现不佳。这是因为基于延迟的拥塞控制算法（如 GCC）可能不会对丢包敏感，除非丢包率很高（丢包率 >10%）。

我们进一步在图4.12中展示了基于 RTP/RTCP 的详细结果，以便更好地理解 Zhuge 的优化。注意 Y 轴都是对数坐标。我们观察到，Zhuge 可以在所有尾部百分位数上减少尾部延迟、长帧延迟比例和低帧率比例。例如，基于数据集 W1，P99 尾部延迟从 400ms 减少到 170ms，400ms 延迟帧比例从 1% 减少到 0.55%。此外，Zhuge 还可以将数据集 W1 和 C1 中的低帧率比例减少至少 50%。

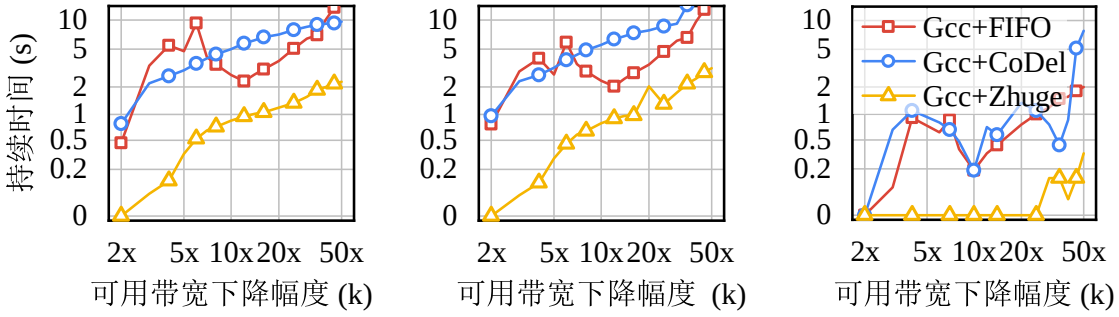
其次是 TCP 协议。如图4.11所示，对于 TCP，Zhuge 可以优于其他的基于接入点的方案（Copa+FastAck）并与端-接入点协同方案（ABC）在所有数据集中实现可比较的性能。在尾部延迟方面，Copa+Zhuge 可以优于 Copa 和 Copa+FastAck。我们还观察到，Copa+FastAck 并不总是比 Copa 表现更好，这是由于 FastAck 的过度重传策略。ABC 在数据集 C3 上的性能优于 Copa+Zhuge，因为 ABC 可以与接入点和端主机协调定制的反馈消息，这可能无法在大规模部署中实现，如 §4.2.3中所述。对于帧延迟，Copa+Zhuge 在所有 trace 中都比竞争对手（包括 ABC）具有

最佳性能，除了 C1，其中 Copa+FastAck 略优于 Copa+Zhuge。ABC 在帧延迟方面表现不佳，这是由于其过度的速率升级策略。我们还在 ABC 论文中使用的数据集上重复我们的实验。我们发现 Zhuge 也能够有和 ABC 可比较的性能。

4.7.4　无线波动下的基准测试

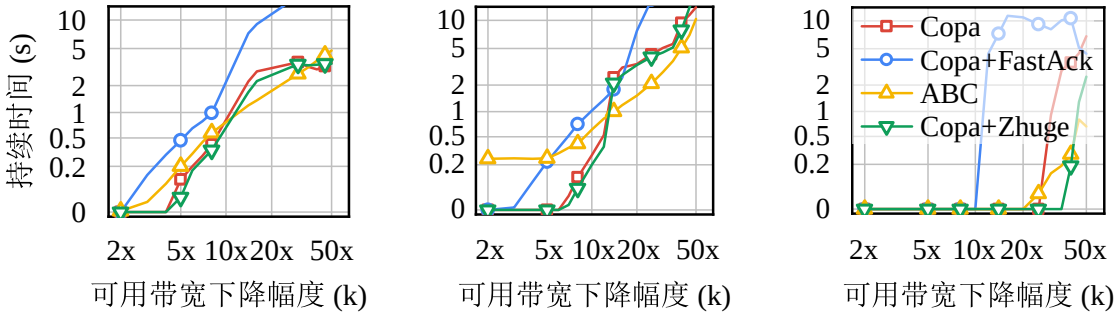
我们进一步仿真了 Zhuge 在带宽降低、流竞争和无线干扰下的性能。

4.7.4.1　带宽下降



(a) 网络 RTT>200ms　(b) 帧延迟 >400ms　(c) 帧率 <10fps

图 4.13　在 RTP 下出现可用带宽下降时的性能比较



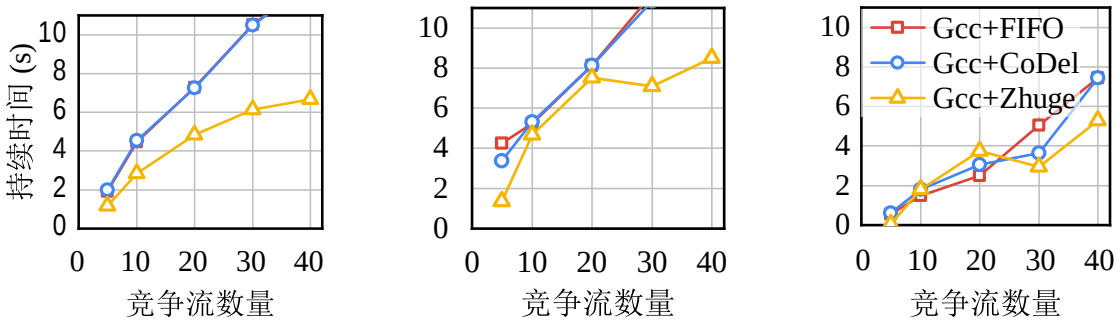
(a) 网络 RTT>200ms　(b) 帧延迟 >400ms　(c) 帧率 <10fps

图 4.14　在 TCP 下出现可用带宽下降时的性能比较

首先，我们评估了带宽下降的场景。我们评估了 Zhuge 在带宽下降时的快速适应能力，并减少了网络条件和应用程序性能下降的时间。我们首先模拟了 50ms RTT 和 30Mbps 带宽的链路。当拥塞控制算法达到稳态时，我们将带宽减少为原来的二分之一到五十分之一（k 值），并测量 RTT>200ms、帧延迟 >400ms 和帧率 <10fps 收敛前的持续时间。如图4.13所示，对于 RTP/RTCP，Gcc+Zhuge 可以减少网络下降和应用程序性能下降的时间至少 50%。TCP 的结果与图4.14中的结果类似。与 Copa 和 Copa+FastAck 的更好表现相比，Copa+Zhuge 可以在 $k < 30$ 时将网络 RTT 的持续时间减少至少 14% 到 64.3%。对于 $k \geqslant 30$，我们的观察是由于严

重的丢包，TCP 重传超时（RTO）主要限制了降级持续时间，Zhuge 的性能提升不如人意。类似地，Zhuge 在 $k < 15$ 时优于 ABC（联合网络主机优化）。根据我们在图4.2中的测量，99% 的带宽下降情况都落在 $k < 15$，Zhuge 带来了很好的改进。

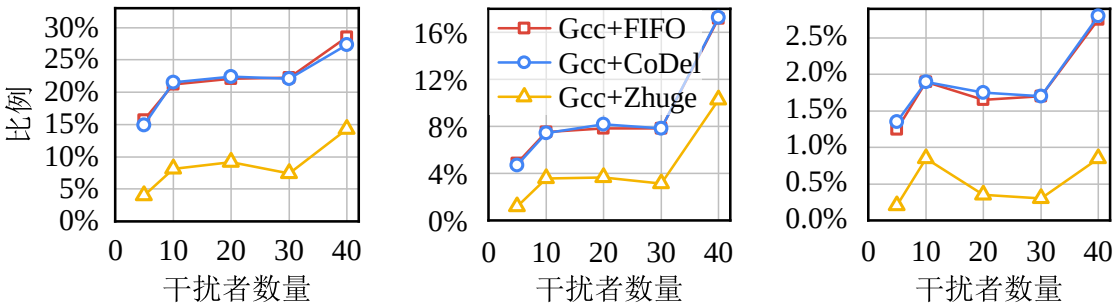
4.7.4.2　流竞争



(a) 网络 RTT>200ms　(b) 帧延迟 >400ms　(c) 帧率 <10fps

图 4.15　在 RTP 下出现流竞争时的性能比较

其次，我们评估了流竞争的场景。我们接下来研究了 Zhuge 在遇到同一瓶颈队列上的竞争者时的表现。我们启动了不同数量的采用 TCP CUBIC 的竞争流量，并让它们在接入点上竞争。我们测量了网络 RTT>200ms、帧延迟 >400ms 和帧率 <10fps 的持续时间。如图4.15所示，与 FIFO 和 CoDel 相比，Zhuge 可以在所有情况下将性能下降的持续时间减少至多 40%。因此，Zhuge 可以有效地减轻竞争下的性能下降。



(a) 网络 RTT　(b) 帧延迟 >400ms　(c) 帧率 <10fps

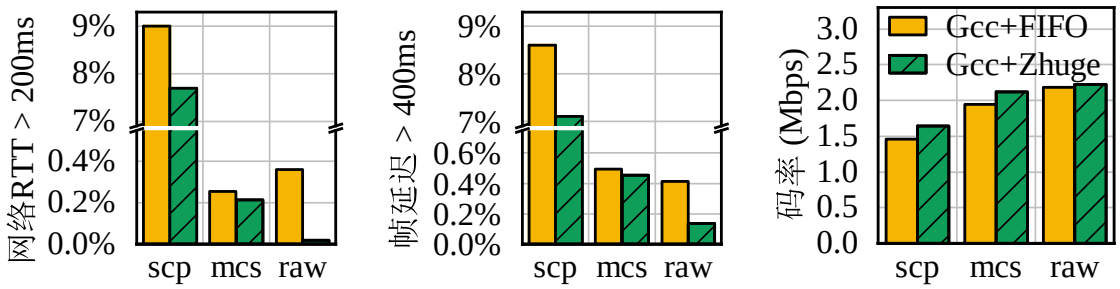
图 4.16　在 RTP 协议中出现无线干扰时的性能比较

4.7.4.3　无线干扰

最后，我们评估了无线干扰的场景。我们测量了不同数量的无线干扰者对 Zhuge 的性能影响。这些干扰者也是基于 TCP CUBIC 的大文件传输应用，但这些应用连接到不同的接入点。它们与 Zhuge 优化的实时多媒体传输流竞争同一无线信道。我们将干扰者的数量从 5 到 40 进行变化。请注意，在无线干扰的场景中，

无线信道中的干扰一直存在，因此我们无法像之前两种情况那样计算单个事件的下降持续时间。如图4.16所示，Zhuge 可以将网络条件和应用程序性能下降的持续时间减少至少 50%。请注意，根据 Cisco 最近的测量[16]，在 2.4GHz 信道上，90% 的干扰者数量最多为 29。因此，Zhuge 可以在嘈杂的无线环境中带来好处。

4.7.5　真实世界实验



(a) 网络 RTT　　(b) 帧延迟　　(c) 码率

图 4.17　基于 OpenWrt 的 WiFi 无线接入点测试平台上的实验评估

我们进一步评估了 Zhuge 在基于 OpenWrt 的 WiFi 无线接入点测试平台上的性能。我们在两台笔记本电脑上的 Microsoft Edge 浏览器中使用 WebRTC 的 API[116]设置了一个实时多媒体的服务器和客户端。服务器通过 RTP/RTCP 和 GCC 将时间戳视频流传输到客户端。服务器与无线接入点通过有线连接，而客户端通过 WiFi 连接到无线接入点。我们评估了 Zhuge 在以下场景下的性能，每个场景持续 6 小时。

- `scp`. 这个实验旨在评估 Zhuge 在实时多媒体流量竞争中的性能，与其他流量竞争。我们每 30 秒启动和停止一次从服务器到客户端的 scp 文件传输。
- `mcs`. 这个实验旨在模拟波动的无线信道。802.11 接入点将在链路层动态地改变调制解调方式（MCS）以适应信道条件。因此，类似于 [78]，我们每 30 秒使用 Linux `iw` 命令随机地改变 MCS，并评估 Zhuge 对波动的反应。
- `raw`. 这里还报告了在一个拥挤的办公室中运行实时多媒体应用程序而不进行任何配置的结果。

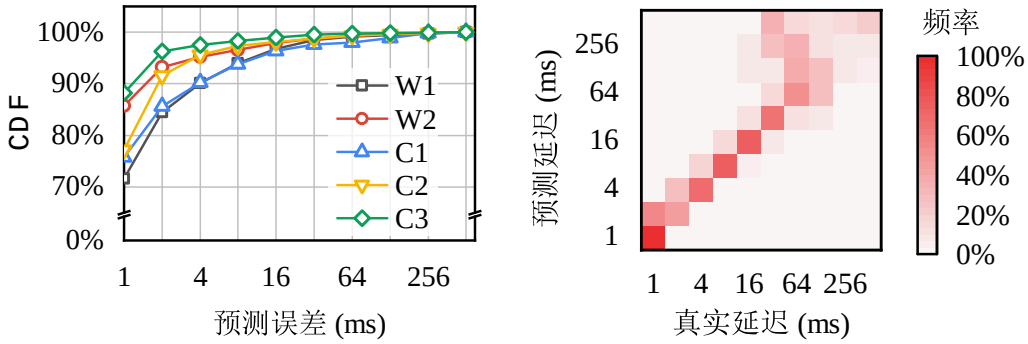
我们通过分析数据包捕获来测量网络 RTT，并通过计算视频发送和视频接收之间的时间戳差来计算帧延迟。如图4.17(a)和4.17(b)所示，Zhuge 的网络 RTT 和帧延迟与基线相比在所有情况下均提高了 17% 至 95%（网络 RTT）和 9% 至 67%（帧延迟）。这表明 Zhuge 可以有效地减少实际无线环境中的尾部延迟。

同时，我们还评估了 Zhuge 在稳定的无线信道下维持与基线相似性能的能力。我们通过测量基于 Microsoft Edge 的视频的平均比特率来评估稳态性能，并在图4.17(c)中呈现结果。我们观察到 Zhuge 可以保持类似的平均比特率，证明它在

稳态中维持性能。请注意，在这里尾部延迟的改进并不会反映在比特率结果中。

4.7.6　深入研究

最后，我们评估了 Zhuge 的公平性和运行时开销。



(a) 仿真数据集上的预测误差.　　(b) 热度图 (按每行归一化).

图 4.18　Zhuge 命运预测器的预测准确性。

4.7.6.1　估计准确性

我们测量了在估计数据包延迟时的准确性（§4.4.1）。我们将估计的延迟与后来测量的相同数据包的实际延迟进行比较。我们在图4.18(a)中展示了不同数据集中预测误差的分布。在我们的实验中，大多数情况下，预测误差都远小于 RTT（50ms）。我们还将不同的结果放入不同的桶中，并在图4.18(b)中呈现每个桶的频率的热度图。如图4.18(b)所示，当估计延迟较低（1-64ms）时，估计通常是准确的。当估计延迟较高（>64ms）时，估计可能不准确，但实际延迟仍然足够高（超过一个 RTT），此时发送方就已经会减少发送速率。

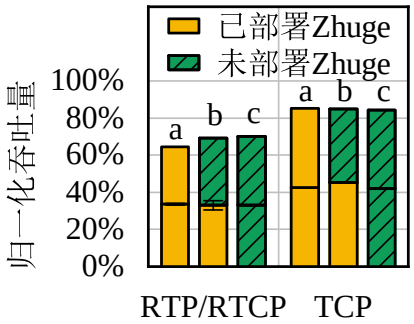


图 4.19　Zhuge 的公平性

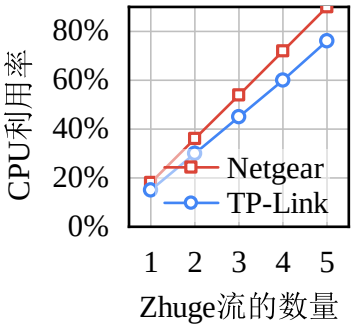


图 4.20　CPU 开销

4.7.6.2　内部公平性。

我们分析了 Zhuge 是否会影响稳态中的比特率公平性，当同时优化两个实时多媒体传输流量的时候。我们报告了两个流量在竞争同一个接入点的时候按归一化为链路容量后的吞吐量。图4.19中的柱状图 *a* 报告了两个流量没有 Zhuge 的吞

吐量，而条形图 *c* 报告了两个流量都由 Zhuge 优化的吞吐量。我们发现，对于 GCC 和 Copa，稳态中的比特率公平性不受 Zhuge 的影响。对于 GCC，Zhuge 甚至略微提高了平均流比特率 10%。这是因为 Zhuge 使发送方能够更快地对发送速率超过链路容量的情况做出反应。

4.7.6.3　外部公平性。

我们评估了 Zhuge 是否会通过在竞争中损害具有相同拥塞控制算法的其他流量来优化流量。我们测量了两个 RTC 流量的比特率，其中一个由 Zhuge 优化，另一个没有。我们在图4.19中的条形图 *b* 中呈现结果。对于 GCC 和 Copa，两个流量的比特率差异都小于 3%。因此，如 §4.6所述，Zhuge 的性能改进不是建立在牺牲其他流量的性能上的。相反，两个流量公平竞争，如拥塞控制算法所期望的那样。

4.7.6.4　运行时开销

Zhuge 也可以通过接受可接受的开销来实现。我们在一个基于 OpenWrt 的 Netgear WiFi 无线接入点和一个 TP-Link TL-WDR4900 无线接入点上测量了 Zhuge 的 CPU 利用率。我们通过 Zhuge 处理不同数量的并发未加密的 RTC 流量来测量 CPU 利用率，并在 Figure 4.20中呈现结果。这两个十年前制造的 AP 仍然可以支持 Zhuge 处理 5 个并发的 RTC 流量，这可以覆盖许多真实场景（例如家庭 WiFi）。

有几个方向可以优化 Zhuge 的资源开销。首先，当 CPU 利用率很高时，Zhuge 可以选择性地更新网络状况，而不是估计所有下行数据包。只要估计之间的时间间隔是微不足道的（例如几毫秒），则控制循环仍然可以减少。此外，我们的 Zhuge 原型仅仅是实现基于用户空间数据包套接字，后续可以进一步通过将 Zhuge 作为内核模块插入来优化。最后，还有一些成功解决方案部署在商业无线接入点中[16,117]。因此，我们对在大规模部署 Zhuge 有进一步优化的信心。

4.8　小结

我们提出了 Zhuge，一个在无线接入点中的解决方案，可以通过减少控制回路以减轻实时多媒体传输应用程序在无线网络中的尾延迟。Zhuge 使用命运预测器预测每个数据包到达时的命运，并使用反馈更新器通过多种协议快速通知发送方这些延迟预测的结果。我们使用真实世界的数据集模拟和在测试基础设施中的部署来评估 Zhuge 的性能。实验表明，Zhuge 在不同场景中将长延迟的尾部和实时多媒体应用程序性能降低减少了 17% 到 95%。

第 5 章　控制通路决策：基于规则地转换控制策略

5.1　本章引言

近年来，深度学习（Deep Learning，DL）已经被广泛应用于网络优化问题，包括视频流媒体[23,118-119]、本地流量控制[120-121]和网络资源管理[122-124]。深度学习的关键启发是利用深度神经网络（Deep Neural Networks，DNNs）来拟合复杂的函数，从而实现预测[125-126]。此外，深度神经网络可以与标准优化技术（如强化学习[127]）结合，从而实现数据驱动和自动化的性能提升。因此，以前的工作已经证明了深度学习在多个网络应用中的显著性能提升，包括手工设计的启发式算法[23,120,128]。

然而，深度神经网络的优异性能来自于其使用数百万甚至数十亿个参数[94,125]。这种成本根本上源于深度神经网络的设计，因为它们通常需要大量的参数来实现通用函数近似[126]。因此，网络运营商必须将深度神经网络视为大型黑盒子[129-130]，这使得基于深度学习的网络系统难以调试、部署和进行临时调整 (§5.2.1)。因此，网络运营商坚持认为，基于深度学习的网络系统在实践中不适合进行关键部署。

数年来，机器学习社区已经开发了几种技术来理解深度神经网络的行为，例如图像识别[131-132]和语言翻译[133-134]。这些技术专注于外科监测神经元的激活，以确定神经元对哪些特征敏感[132]。然而，直接将这些技术应用于基于深度学习的网络系统是不合适的—网络运营商通常希望从输入（例如，将具有某些头部的数据包调度到端口）映射到简单、确定性的控制规则，而不是对深度神经网络的操作细节进行挑剔。此外，网络系统在应用设置和输入数据结构方面都是多样化的。当前的深度神经网络解释工具，主要针对结构良好的向量输入（例如，图像、句子），在多样化的网络系统中都不够充分。因此，需要为网络系统开发一个特定于网络系统的可解释深度学习框架，以便网络运营商可以在网络系统中使用深度学习，同时保持对系统行为的控制权。

本章中，我们的高层次设计目标是通过人类可读的控制策略来解释基于深度学习的网络系统，以便网络运营商可以轻松地调试、部署和临时调整基于深度学习的网络系统。本章提出了 Metis[①]来提供可解释性。为了支持各种网络系统，Metis 发现大多数视频流媒体的自适应系统是**本地系统**，它们在本地收集信息并仅为一个实例做出决策，例如端设备上的拥塞控制代理和交换机上的流调度器。

① Metis 是希腊神话中的一位女神，起着一种咨询的角色。

具体来说，我们采用了一种决策树转换方法[135-136]来转换这些系统。这种设计选择的主要观察是，现有的启发式实时多媒体传输系统通常是**规则**决策系统 (§5.3.1)，它们具有相当简单的决策逻辑（例如，基于缓冲区大小来选择速率的方案[22]）。因此，本工作的转换是建立在教师-学生的模仿学习训练过程之上的，其中深度神经网络策略充当教师并生成输入输出样本来构建学生决策树[136]。然而，为了与深度神经网络匹配性能，传统的决策树算法[137]通常会输出大量的分支，这些分支实际上是不可解释的。我们利用两个重要的观察结果来将分支修剪到网络运营商可以接受的数量。首先，本地系统中的合理策略通常会在观察到的状态中一致地输出相同的控制动作。例如，任何性能良好的自适应码率系统都会在缓冲区充足的情况下输出相同的比特率。因此，通过依赖教师深度神经网络生成的数据，决策树可以轻松地剪掉决策空间。第二，不同的输入输出对对策的性能贡献是不同的。我们采用一种特殊的重采样方法[135]，使得教师深度神经网络可以指导决策树优先考虑导致最佳结果的行动。从经验上看，我们的决策树可以生成人类可读的解释 (§5.5.1)，而且性能损失可以控制在 2% 以内。

为了完整评估 Metis 的可解释性，我们使用 Metis 生成基于深度学习的自适应码率系统的可解释策略 (§5.5.1)。例如，我们解释了 Pensieve[23]的比特率自适应策略，并提出了一个新的决策变量。我们还展示了 Metis 在基于深度学习的网络系统的设计、调试、部署中的三种用例。(i) Metis 帮助网络运营商使用 Pensieve 的神经网络结构来将平均 QoE 改善了 5.1% (§5.5.3)。(ii) Metis 调试 Pensieve 的深度神经网络，并在不使用决策树的情况下将平均 QoE 提高了 4% (§5.5.3)。(iii) Metis 使 Pensieve 的轻量级版本具有更短的决策延迟，最高是 156 分之一 (§5.5.5)。

本章主要贡献如下：

- Metis，一个通用的框架，用于解释基于深度学习的网络系统，以便网络运营商可以轻松地调试、部署和临时调整基于深度学习的网络系统。它通过决策树来解释实时多媒体传输的决策系统（§5.3）。
- Metis 的原型实现，包括基于深度学习的多媒体传输码率决策系统 Pensieve[23](§5.4)，以及它们的解释，包括捕获已知的启发式策略和发现新的知识 (§5.5.1)。
- 三个 Metis 的用例，用于帮助网络运营商设计 (§5.5.3)，调试 (§5.5.4)，和部署 (§5.5.5) 基于深度学习的网络系统。

据本文所知，Metis 是第一个用于解释基于深度学习的网络系统的通用框架，并适用于实时多媒体传输应用。Metis 源代码可在https://github.com/transys-project/metis/上获得。我们相信 Metis 将通过缩短、稳定网络算法的决策来加速基于深度

学习的网络系统在实践中的部署。

5.2　动机

我们通过分析下面两个问题来引入 Metis。

1. 为什么当前的基于深度学习的网络系统缺乏可解释性？（§5.2.1）
2. 为什么现有的解释方法对基于深度学习的网络系统不起作用？（§5.2.2）

5.2.1　现有系统的缺陷

深度神经网络的黑盒子属性使得网络运营商难以理解网络系统的决策。在不能理解决策的情况下，网络运营商可能没有足够的信心将基于深度学习的网络系统部署到实际网络中[130]。此外，如图 5.1所示，基于以下原因，黑盒子属性会给网络系统的调试、在线部署和临时调整带来很多不便。

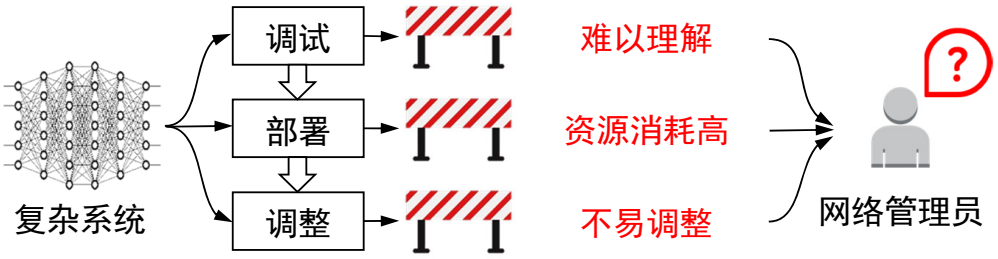


图 5.1　复杂决策系统在网络系统的生命周期中许多环节都会制造障碍

（1）难以理解的结构。深度神经网络可能包含数千到数十亿个神经元[94]，使得它们对网络运营商来说难以理解。由于深度神经网络的复杂结构，当基于深度学习的网络系统无法按预期执行时，网络运营商将难以定位错误的组件。即使在找到深度神经网络结构中的次优设计之后，网络运营商也面临着重新设计它们以获得更好性能的挑战。如果网络运营商能够跟踪输入和输出之间的映射函数，那么就会更容易调试和改进基于深度学习的网络系统。

（2）重量级的部署。深度神经网络在资源消耗和决策延迟方面都是笨重的[138]。即使使用高级硬件（例如 GPU），深度神经网络也可能需要几十毫秒的决策时间 (§5.5.5)。相比之下，网络系统，尤其是端设备（例如移动电话）上的实时多媒体传输系统，都是资源受限且延迟敏感的[138]。例如，将基于深度神经网络的自适应码率算法加载到移动客户端上可能会使页面加载时间增加大约 10 秒 (§5.5.5)，这将使用户离开该页面。现有的系统通常只提供“尽力而为”的服务，并在无法满足资源和延迟约束时回滚到启发式策略[120]，从而降低了深度神经网络的性能。

（3）不可调整的策略。实际部署网络系统还需要临时调整或添加临时功能。例如，我们可以调整公平调度中不同作业的权重，以跟上工作负载的波动[128]。但是，由于缺乏解释，网络运营商在需要调整网络系统时会遇到困难。如果网络运营商不理解深度神经网络做出这样的决定的原因，那么任意调整可能会导致严重的性能下降。例如，当网络运营商想要手动将流从路径上重新路由时，如果没有决策的解释，网络运营商可能不知道如何以及在哪里放置该流。

讨论。应用深度神经网络到网络系统的工作仍处于初级阶段：Pensieve[23]、AuTO[120]和 RouteNet[122]（分别于 2017、2018 和 2019 年发表）的深度神经网络层数都不到十层。与此相比，其他领域中深度神经网络层数的增长已经呈指数级增长（见图 5.2，图像来源于[139]）。最近的一些自然语言处理的大模型（如 GPT）甚至包含数十亿个参数[94]。虽然我们并不是说越大越好，但毫无疑问，更大的深度神经网络会加剧问题并阻碍将基于深度学习的网络系统部署到实际网络中。

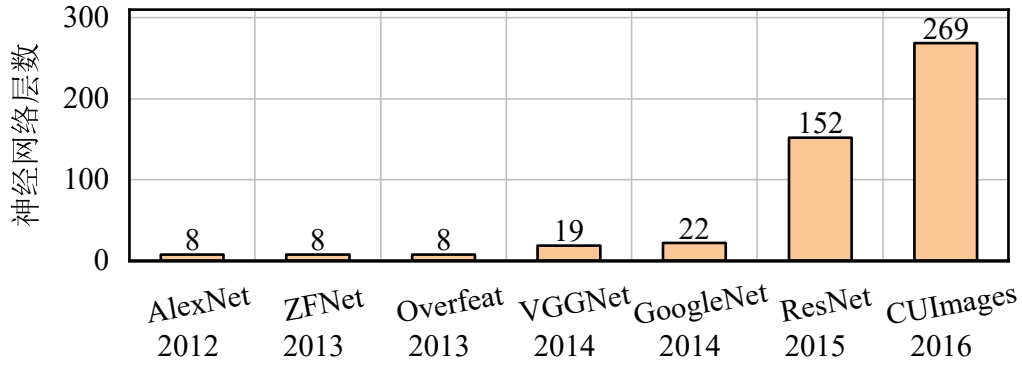


图 5.2　ImageNet 挑战获胜者中深度神经网络的指数增长[140]

5.2.2　现有解释方法的不足

对于基于深度学习的网络系统，现有的解释方法[141-142]在以下方面不足：

（1）不同的解释目标。为什么深度神经网络做出某个决定的问题可能从两个角度得到答案。在机器学习社区，答案可能是理解机器学习模型的**内部机制**（例如，哪些神经元对某些特定输入特征有所反应）[131-132]。这就像是试图通过手术理解大脑的工作原理。相比之下，网络运营商期望的答案是**输入和输出之间的关系**（例如，哪些输入特征影响决策）[130]。网络运营商需要的是一种方法来解释深度神经网络输入和输出之间的映射。

（2）多样化的网络系统。基于深度学习的网络系统具有不同的应用场景，并基于各种深度学习方法，例如前馈神经网络（FNN）[23]、循环神经网络（RNN）[143]和图神经网络（GNN）[128]。因此，使用单一的解释方法来解释多样化的基于深度学习的网络系统是不够的。例如，LEMNA[144]只能解释 RNN 的行为，因此不

适用于基于 GNN 的网络系统[128]。Metis 观察到实时多媒体传输的速率调整机制其实是本地系统，并开发了相应的技术。

因此，Metis 引入了基于决策树的方法以解释基于深度学习的网络系统。

5.3　决策树解释

在本节中，我们首先介绍决策树解释方法的设计考量 (§5.3.1)，然后介绍将深度神经网络转换为决策树的详细方法 (§5.3.2)。

5.3.1　设计考量：决策树

如 §5.1所述，Metis 将深度神经网络转换为决策树，以便解释网络系统的行为。在这一过程中，可能有很多候选模型，例如（超）线性回归[144-145]、决策树[135-136]等。我们推荐读者参考[141-142]以有一个更全面、深入的理解。

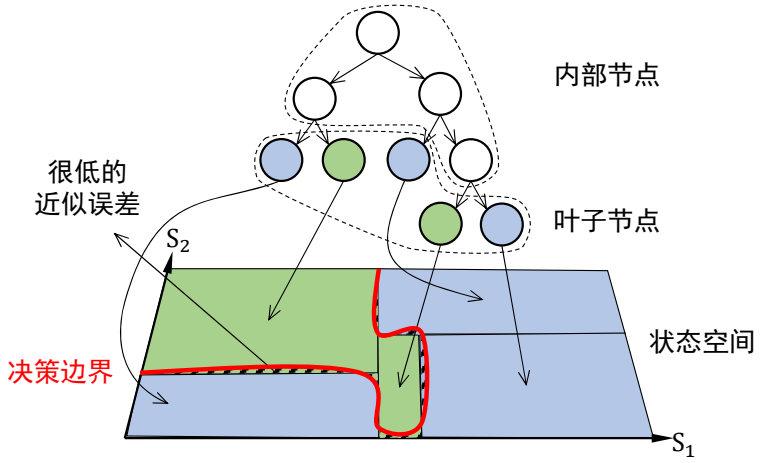


图 5.3　决策树模仿原有决策边界的示意图

在本章中，我们选择决策树作为解释方法的基础是基于以下三个原因：首先，决策树的逻辑结构与网络系统的策略相似，例如，交换机的流调度算法通常依赖于一组转发规则，例如最短作业优先[84]。自适应码率算法依赖于基于缓冲区占用和预测吞吐量的预先计算规则[34,38]。其次，如图5.3所示，决策树具有丰富的表达能力和高度的可信度，因为它们是非参数的，可以表示非常复杂的策略[146]。我们在[147]中展示了决策树转换的性能与其他方法[144-145]的比较。第三，决策树对于网络系统来说是轻量级的，这将为资源消耗和决策延迟带来进一步的好处 (§5.5.5)。还有一些研究工作尝试将深度神经网络解释为编程语言[148-149]。但是，为每个网络系统设计不同的原语是耗时且低效的。

有了决策树形式的解释结果，决策过程就变得透明了。我们便可以解释理解

决策结果。同时，我们也可以调试深度神经网络模型，以便在它们生成次优决策时进行调试 (§5.5.4)。此外，由于决策树的大小更小，计算成本更低，因此我们还可以在线部署决策树，而不是部署重量级的深度神经网络模型。这将导致低决策延迟和资源消耗 (§5.5.5)。

5.3.2　转化方法

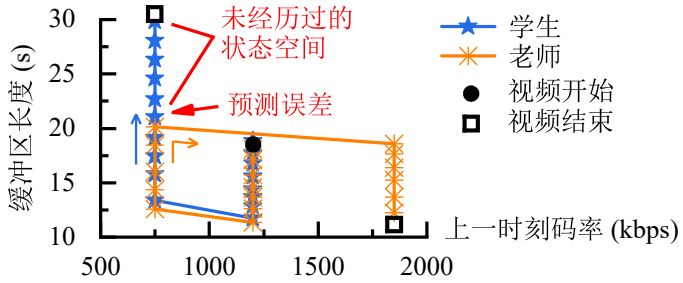


图 5.4　老师纠正学生决策的示意图

为了从一个精调的深度神经网络中提取决策树，我们采用了 [135] 中提出的教师-学生模仿训练方法。如果不采用教师-学生模仿训练方法，一个错误的决策便可能将学生带离老师原有的轨迹。如图5.4所示，一个错误的决策可能会将原有决策树带入未经历过的一片状态空间。决策树因为在这片区域内没有任何先验知识，它便可能会做出错误决策。这些错误决策会进一步将决策树带着偏离原有轨迹，进一步恶化性能。因此，本工作持续仿真决策树的性能，并让原有的自适应码率算法作为老师持续纠正决策树作为学生做出的决策。决策树便会因此逐渐学到如何在整个状态空间中决策。

我们按照如下步骤重现网络系统的关键转换步骤：

步骤 1：跟踪收集。训练决策树时，重要的是从深度神经网络中获取适当的数据集。简单地覆盖所有可能的（状态，动作）对是太昂贵的，而且不能充分反映目标策略的状态分布。因此，Metis 跟踪由教师深度神经网络生成的轨迹。此外，网络系统是顺序决策过程，其中每个动作对未来状态都有长期影响。因此，决策树可能会由于不完美的转换而与深度神经网络轨迹产生显着偏差[135]。为了使转换的策略更加健壮，我们让深度神经网络策略接管偏离轨迹的控制，并重新收集（状态，动作）对以改进转换训练。我们重复该过程，直到偏差被限制在可接受范围内（即，转换的策略紧跟深度神经网络轨迹）。

步骤 2：重采样。本地系统通常优化**策略**而不是独立的动作[23,120-121]。在这种情况下，网络系统的不同动作可能对优化目标的重要性不同。例如，ABR 算法在缓冲区极低时下载大块会导致长时间停顿，从而导致严重的性能下降。同时，当

网络条件和缓冲区处于适中时下载稍大的块不会产生显着影响。然而，决策树算法旨在优化单个动作的准确性，而将所有动作视为相同。因此，它们的优化目标不匹配。现有的基于深度学习的本地系统采用强化学习（RL）来优化策略而不是单个动作，其中每个（状态，动作）的**优势**表示对优化目标的重要性。因此，我们遵循最近在 RL 策略中将深度神经网络转换为决策树的进展[135]，并根据优势函数对 $\mathcal{D}$ 进行重采样。对于每对 (s,a)，我们计算其优势函数 $A(s,a)$，并将其作为采样概率的权重。

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

其中，$V^{(\pi^*)}(s)$ 和 $Q^{(\pi^*)}(s,a)$ 是 RL 中的价值函数和 Q 函数[127]。价值函数表示从状态 s 开始并遵循策略 π 的预期总奖励。Q 函数进一步指定下一步动作 a。π^* 是深度神经网络策略，A 是动作空间。$\mathbb{1}(x)$ 是指示函数，当且仅当 x 为真时等于 1。我们将公式5.1的分析细节放在[147]中。

然后，我们在重采样的数据集上重新训练决策树。我们的实验结果表明，重采样步骤可以提高 73% 的数据集中的 QoE。

步骤 3：修剪。由于决策树的大小有时比网络运营商能够理解的更大，我们采用成本复杂度修剪（CCP）[137]来根据网络运营商的要求减少分支的数量。与其他修剪方法相比，CCP 在实践中可以获得与相似错误率相似的较小决策树[150]。在其核心，CCP 创建了一个修剪后决策树的复杂度成本函数，以平衡准确性和复杂性。此外，对于网络系统中的连续输出（例如，队列阈值[120]），我们采用**回归树**的设计来生成实值输出[151]。在我们的实验中，对于 Pensieve，叶节点的大小可能达到 1000。使用 CCP，修剪决策树到 200 个叶节点只会导致性能下降不到 1%。

步骤 4：部署。最后，网络运营商可以在线部署转换后的模型，并同时享受深度学习带来的性能提升和转换后模型提供的可解释性。我们的评估表明，两个基于深度学习的网络系统的决策树的性能下降不超过 2%（§5.5.5）。我们还在 §5.5.4和§5.5.5中介绍了将深度学习网络系统转换为决策树的其他好处（易于调试和轻量级部署）。

5.4　系统实现

在现有互联网视频传输中，每个视频由许多**块**（几秒钟的播放时间）组成，每个块以多个比特率编码[23]。Pensieve 是一种基于深度强化学习的用于优化比特率和网络观测值的自适应码率系统（例如，过去的块吞吐量和缓冲区占用）。

我们在本节中使用 Pensieve 的实现[23]。除非另有说明，我们使用相同的视频。视频块大小和比特率分别设置为 4 秒和 {300, 750, 1200, 1850, 2850, 4300} kbps。真实网络数据集包括 250 个 HSDPA 数据集的记录[152]和 205 个 FCC 数据集的记录[153]。我们使用 `tf.js`[154]将深度神经网络集成到 JavaScript 中，以便在浏览器中运行 Pensieve。我们采用了与 Pensieve 相同的环境和 QoE 指标。

我们接着实现了 Metis+Pensieve。我们使用 [23] 提供的精调后的模型来生成决策树。我们将五个基线自适应码率算法（BB[22]，RB[23]，Festive[155]，BOLA[34]，rMPC[38]）作为基线算法来对比，并将其迁移到 `dash.js`[156]中进行测试。

5.5　实验评估

本节经验性地评估了 Metis 的可解释性。我们首先评估 Metis 在两种类型的基于深度学习的网络系统上的可解释性。然后，我们展示了 Metis 如何解决现有的基于深度学习的网络系统的缺点（§5.2.1）。最后，我们对 Metis 的可解释性进行了基准测试。总体而言，我们的实验涵盖了以下方面：

- **系统解释**。我们通过展示 Metis 对 Pensieve 的解释及新发现的知识来证明 Metis 的有效性 (§5.5.1)。
- **模型设计指南**。我们基于 Metis 提供的解释，展示了如何改进 Pensieve 的深度神经网络结构以获得更好的性能 (§5.5.3)。
- **启用可调试性**。我们基于 Pensieve 的一个用例，展示了 Metis 如何调试问题并通过调整决策树的结构来提高 Pensieve 的性能 (§5.5.4)。
- **轻量级部署**。对于 Pensieve，网络运营商可以直接在线部署 Metis 提供的转换后的决策树，并同时获得轻量级部署带来的好处 (§5.5.5)。
- **Metis 深入探索**。我们最后在不同设置下评估了 Metis 的解释性能、参数敏感性和计算开销 (§5.5.1)。

5.5.1　系统解释

有了 Metis，我们可以解释基于深度学习的网络系统的决策过程。我们展示了 Metis 如何解释 Pensieve 的决策过程。

我们在图5.5中展示了 Metis+Pensieve 的前四层决策树。每个节点的决策变量包括上一个块的比特率（$\boldsymbol{r}^t$）、上一个吞吐量（$\boldsymbol{\theta}^t$）、缓冲区占用（$\boldsymbol{B}$）和上一个块的下载时间（$\boldsymbol{T}_t$）。由于我们只展示了决策树的前四层，因此我们用图5.5中的调色板表示每个节点的最终决策的频率。颜色代表该节点选择某个比特率的频率。例如，调色盘中的箭头代表 67% 经过该节点的状态最终选择了 4300kbps，33% 选择

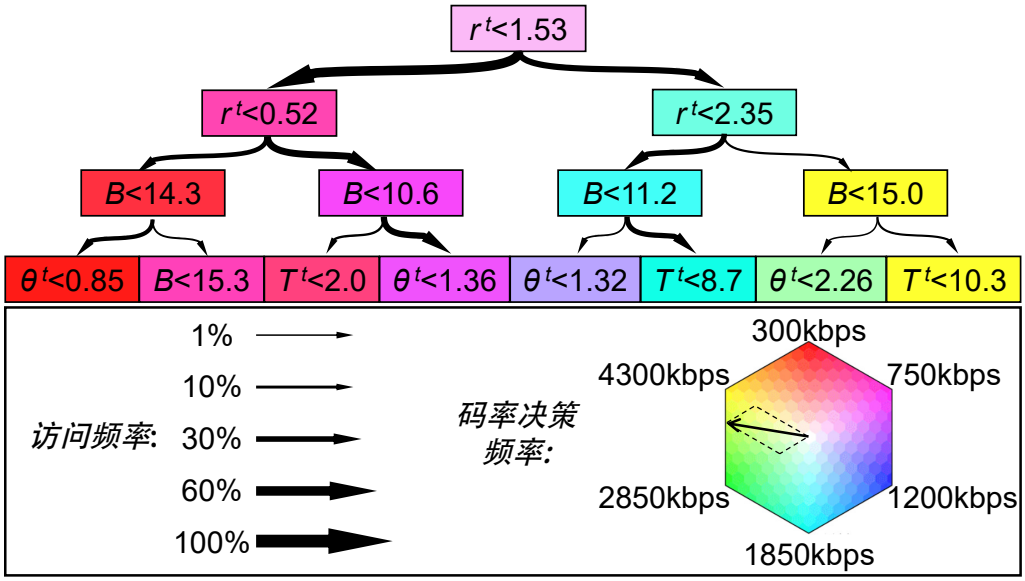


图 5.5　Metis+Pensieve 的决策树的前四层

了 2850kbps。

从图5.5中的解释可以看出，Metis+Pensieve 在两个方面表现出优势。(i) **发现新知识**。在前两层中，Metis+Pensieve 首先根据**上一个块的比特率**将输入分类为四个分支，这与现有方法不同。上一个比特率的选择包含了对输出 QoE 的影响。基于这一观察，我们建议网络运营商可以通过特别关注上一个块的比特率来改进 ABR 算法。我们在 §5.5.3中展示了如何利用这一观察来改进深度神经网络结构。(ii) **捕捉现有的启发式方法**。与现有方法类似，Metis+Pensieve 基于缓冲区占用[22,34]和预测吞吐量[38,156]来做出决策。通过 Metis 的解释，我们可以看到这些启发式方法的影响。例如，我们可以看到 Metis+Pensieve 在缓冲区占用较低时选择了较低的比特率，这与现有方法的启发式方法一致。我们在 §5.5.4中展示了如何利用这一观察来调试 Pensieve 的问题。

5.5.2　性能保持

我们通过对比原有决策算法和 Metis 生成的决策树的 QoE 来评估 Metis 的性能维持性。我们因此测量了 Metis 生成的决策树和原有决策算法的 QoE 比值。QoE 比值小于 100% 表示性能下降。由于 QoE 的值跨越了正负值，我们将所有的 QoE 值归一化到均值为 1，标准差为 1 的分布中。我们在不同的 trace 和不同的 QoE 指标上展示了详细的结果，如图5.6所示。我们展示了三种算法在三种 QoE 指标、七种数据集和三种自适应码率算法上的第 10、25、50、75 和 90 百分位数的 QoE 比值。我们可以看到，大多数中位性能下降小于 5%，这表明 Metis 可以准确地转换复杂的算法在广泛的场景中。

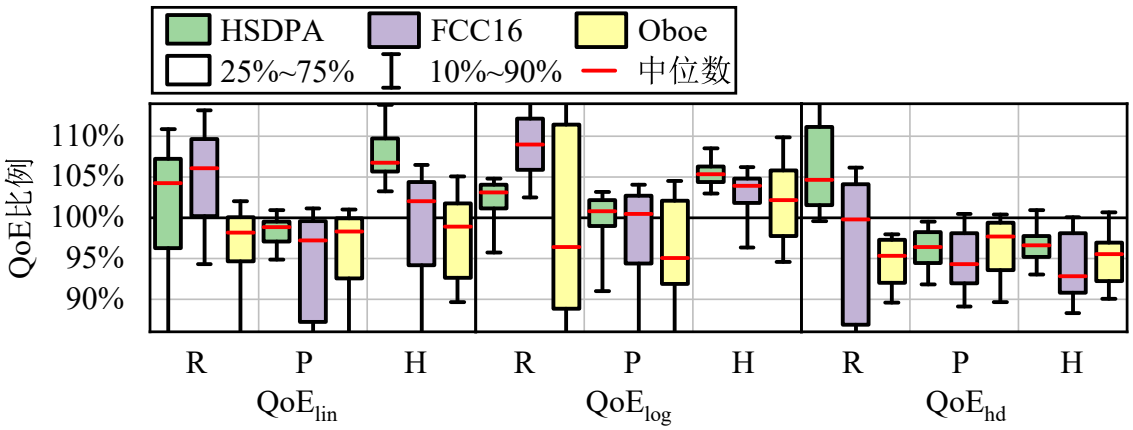
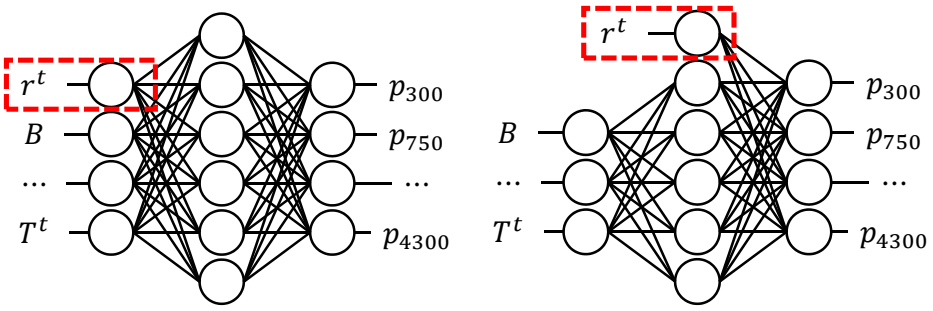


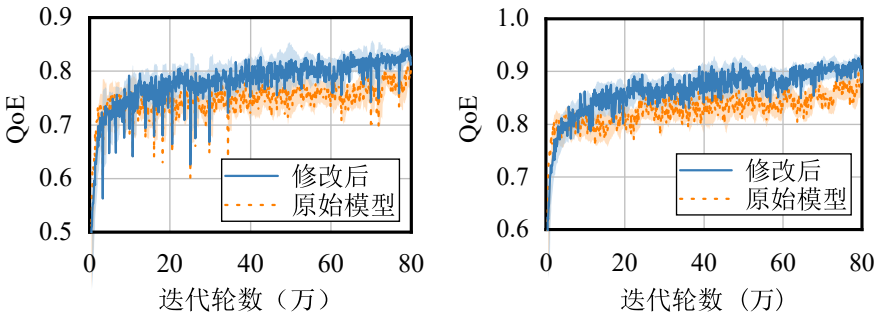
图 5.6　不同 QoE 指标与自适应码率算法下的 QoE 比例

5.5.3　模型设计指导



(a) 原始结构　　(b) 修改后的结构

图 5.7　修改前后的 Pensieve 的深度神经网络结构



(a) 训练集　　(b) 测试集

图 5.8　修改前后的 Pensieve 的 QoE 和训练效率

我们展示了如何利用 Metis 的解释来改进 Pensieve 的深度神经网络结构的一个例子。我们在 §5.5.1中发现，Pensieve 在决策时显著依赖上一个块的比特率（r^t）。这表明 r^t 可能包含对优化结果的重要信息。

为了利用这一观察，我们修改了 Pensieve 的深度神经网络结构，使得 r^t 对输出结果的影响更大。如图5.7(b)所示，我们直接将 r^t 连接到输出层，这样可以更直

接地影响预测结果。我们基于 §5.5.1中的解释，修改了 Pensieve 的深度神经网络结构。虽然两个深度神经网络结构在数学上是等价的，但由于深度神经网络的巨大搜索空间[157]，它们将导致不同的优化性能和训练效率。通过将重要的输入特征放在输出附近（从而简化重要特征与结果之间的关系），修改后的深度神经网络将更多地关注重要特征。

我们在同一训练集上训练了两个深度神经网络模型，并在测试集上评估了它们的 QoE。结果如图5.8所示，阴影区域表示 ± 标准差。从原始模型和修改后的模型的曲线中，我们可以看到修改后的深度神经网络可以提高训练速度和最终 QoE 两个指标。例如，在测试集上，修改后的深度神经网络平均比原始深度神经网络提高了 5.1% 的 QoE①。考虑到视频提供商的观看规模（每天观看数百万小时的视频[158]），即使是小的 QoE 提高也是重要的[119]。此外，修改后的深度神经网络可以在平均情况下节省 55 万轮迭代来达到相同的 QoE，这可以节省 23 小时的时间。

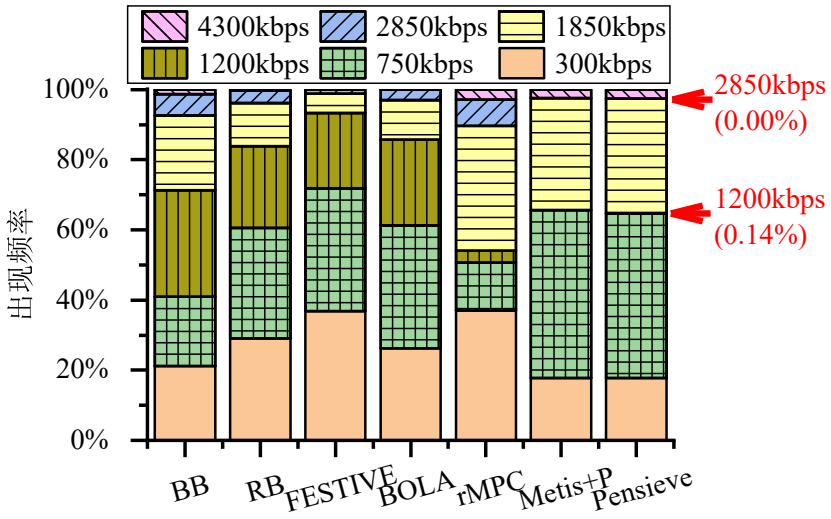
5.5.4　为可调试性赋能

当解释 Pensieve 时，就像 Dethise et al.[129]所报告的那样，我们发现有些比特率很少被 Pensieve 选择。§5.5.1中的实验中选择的比特率的频率如图5.9(a)和5.9(b)所示。Metis+Pensieve 生成的结果与 Pensieve 几乎相同，其中 1200kbps 和 2850kbps 很少被选择。在 300kbps 到 4300kbps 的六个比特率中，两个比特率（1200kbps 和 2850kbps）很少被 Pensieve 选择。这种不平衡引起了我们的兴趣，因为没有被选择的比特率是中位数比特率：最高或最低比特率可能由于网络条件而不被选择，但中位数不应这样。

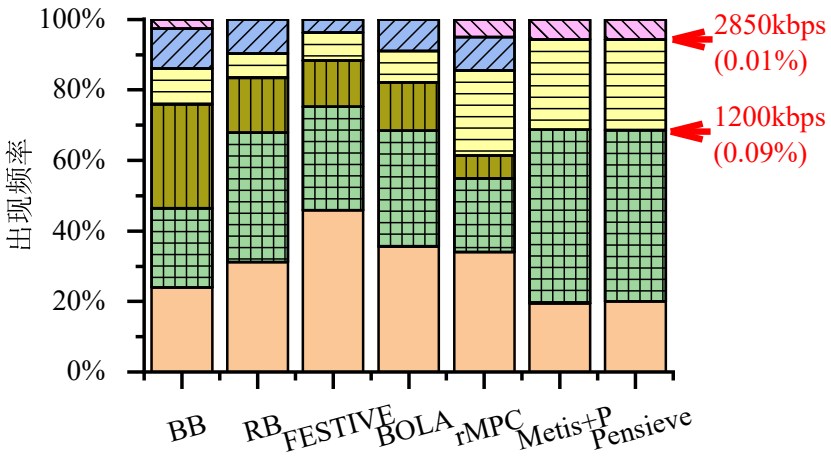
为了进一步探索原因，我们在 300kbps 到 4500kbps 的一组固定带宽上模拟 Pensieve。由于 [23] 使用的示例视频太短，我们将测试视频替换为 1000 秒的视频，并保持所有其他配置与原始实验相同。如图5.9(c)所示，1200kbps 和 2850kbps 仍然不被 Pensieve 首选。例如，在 3000kbps 的链路上②，最优的决策是选择 2850kbps，然而，在这种情况下，Pensieve 选择 2850kbps 的比例仅仅为 0.4%，而其余决策则在 1850kbps 和 4300kbps 之间分配。如图5.10所示，在一条 3000kbps 的链路上，BB，RB 和 rMPC 学习到最优策略并收敛到 2850kbps。Metis+Pensieve (Metis+P) 和 Pensieve 在 1850kbps 和 4300kbps 之间振荡，降低了 QoE。与此相反，其他基线学习到了最优策略并将其决策固定为 2850kbps，从而获得更高的 QoE。类似的观察也可以在 1200kbps 的链路上观察到。

研究了 Pensieve 的原始输出后，我们发现 Pensieve 没有足够的信心来选择任

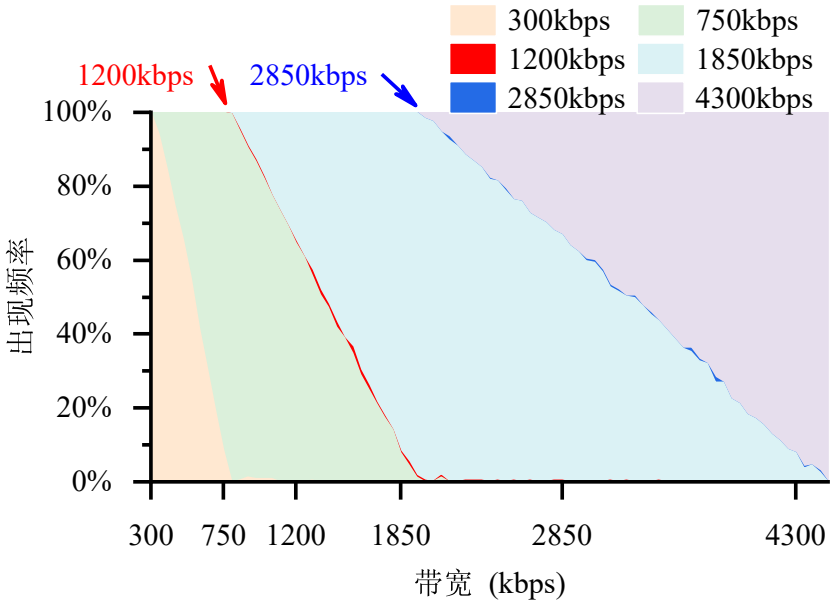
① 注意到，Pensieve 在 [23] 中报告的离线最优性差也只有 9.6%-14.3%。
② 在这种情况下，比特率大约为 2850kbps。



(a) HSDPA 数据集 (12250 个动作)



(b) FCC 数据集 (10045 个动作)



(c) Pensieve 在固定带宽下的性能

图 5.9　Pensieve 在不同场景下的动作选择分布

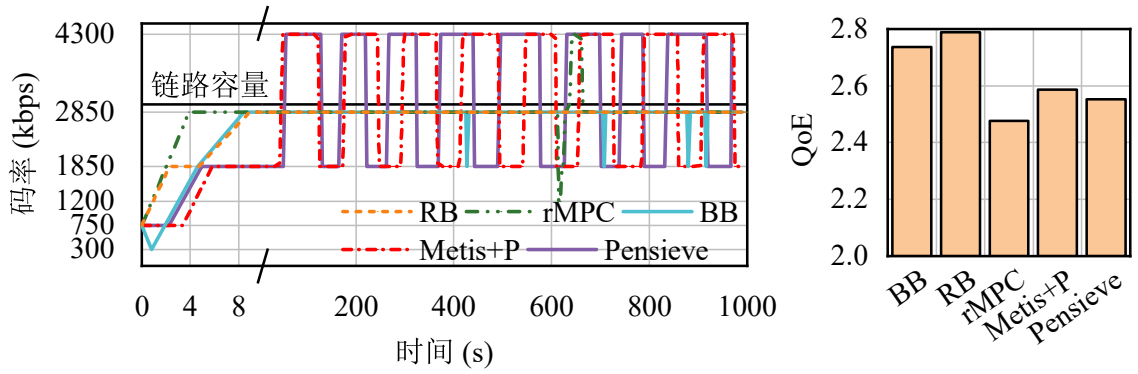


图 5.10　一条 3000kbps 链路上各算法的决策结果

何一个比特率。因此，它在这两个比特率之间振荡。Pensieve 的训练机制可能导致这个问题。在每一步中，深度学习代理尝试强化导致更大奖励的特定行为。在这种情况下，当代理发现六个动作中的四个可以获得相对较好的奖励时，它将通过不断选择这些动作来不断强化这一发现，并最终放弃其他动作。选择更少的动作可以使代理对其具有更高的信心，但也会使代理收敛到这种情况下的局部最优。

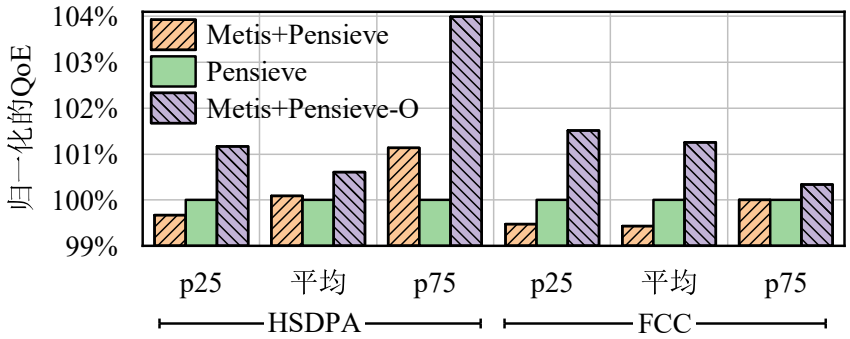


图 5.11　过采样前后算法的性能对比

除了像 Dethise et al.[129]那样发现问题之外，Metis 还可以帮助解决问题。没有 Metis，由于 Pensieve 是基于强化学习设计的，网络运营商没有深度神经网络模型的显式结果。网络运营商可能需要在奖励中惩罚比特率的不平衡，并重新训练深度神经网络模型数小时到数天，而不知道这一深度学习是否可以逃离局部最优。使用 Metis，从深度神经网络到决策树的转换提供了一个网络运营商调试模型的接口。由于训练决策树的数据集 $\mathcal{D}$ 非常不平衡，作为一个简单的解决方案，我们对丢失的比特率进行过采样，以确保采样后的频率大约为 1%。如图5.11所示，在 HSDPA 跟踪上，过采样的决策树（Metis+Pensieve-O）平均比深度神经网络提高 1%，在第 75 百分位上提高 4%。在这里的图中，QoE 被 Pensieve 标准化了。

5.5.5　轻量级部署

Metis 提供的决策树部署起来非常轻量。前面第5.5.2节已经证明，该决策树和原始决策算法之间的性能损失很小（小于 2%）。因此，直接部署 Metis 的决策树将（i）缩短决策延迟，（ii）减少资源消耗并带来更多的性能收益，并且（iii）使其能够部署到高级设备上。

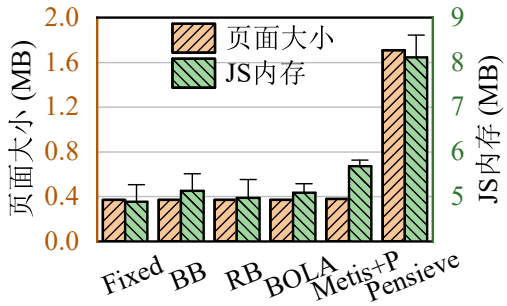


图 5.12　页面大小和 JS 内存

资源消耗。我们评估 Metis+Pensieve 的资源消耗（具体而言，**页面加载时间**和**内存消耗**）。为了消除 DASH 播放器中其他模块的影响，我们将这些 ABR 算法与 `fixed` 算法进行比较，该算法始终选择最低比特率。

对页面加载时间而言，如果 HTML 页面大小太大，用户必须等待很长时间才能开始播放视频。如图5.12所示，`fixed`，BB，RB 和 BOLA 的页面大小几乎相同，因为它们的处理逻辑很简单。Pensieve 增加了 1370KB 的页面大小，因为它需要先下载深度神经网络模型。相比之下，Metis+Pensieve 具有与启发式算法相似的页面大小。当良好的传输速率为 1200kbps（Pensieve 评估跟踪的平均带宽）时，与 `fixed` 相比，ABR 算法的**附加**页面加载时间减少了 156×：Pensieve 引入了 9.36 秒的附加页面加载时间，而 Metis+Penseve 只增加了 60ms。

我们接下来测量了运行时内存，并在图5.12中呈现了结果。由于神经网络的前向传播复杂，Pensieve 消耗了比其他 ABR 算法更多的内存。相比于原始 Pensieve 模型，Metis+Pensieve 引入的额外内存平均减少了 4.0×，峰值减少了 6.6×，这与其他启发式算法的水平相当。

5.5.6　Metis 深入探索

最终，我们概述了深入探索 Metis 的实验。我们将详细的实验设置和更多的实验结果放到 [147] 中。

（1）解释基线比较。我们将 Metis 中的决策树的性能与深度学习社区中的两个基线进行比较。我们实现了 LIME[145]，深度学习社区中最典型的黑盒解释方法

之一，以及 LEMNA[144]，专门为 RNN 中的时间序列设计的解释方法。我们测量了三种解释方法的错误率和误差。基于 Metis 的方法在两个系统上的错误率平均减少了 1.2×-1.7×，在两个基线上的错误率平均减少了 1.2×-1.7×。RMSEs 减少了 1.2×-3.2×。实验在[147]中详细介绍。决策树优于其他两种解释方法，这证实了我们在 §5.3.1中的设计选择。

（2）敏感性分析。我们在 [147] 中测试了 Metis 中的超参数的鲁棒性。对于决策树解释，我们测试了叶节点数的鲁棒性。结果表明，从 10 到 5000 的各种设置（准确度变化在 10% 之内）对于 Pensieve 都表现良好。

（3）计算开销。在 [147] 中，我们的评估表明，将微调的深度神经网络转换为决策树的 Pensieve 和 AuTO 需要不到 40 秒的时间，其实是很快的。

5.6　讨论

本节中，我们讨论了 Metis 的设计选择，泛化能力，局限性和未来的方向。

（1）为何不直接训练决策树？如5.5.5中所示，转换后的决策树在性能上与更大的模型相当。但是，**直接**从头开始训练更简单的模型很难达到相同的性能。本章猜想第一个原因是决策树是非参数模型，不是为连续参数更新和结构调整而设计的。即使有最近的决策树调整技术[159]，高效的调整也依赖于大量的训练数据。另一个可能的解释是**彩票假说**[160-161]：训练深度模型类似于通过购买大量彩票（即构建大型神经网络）来赢得彩票。但是，我们无法提前知道中奖的彩票配置。因此，直接训练更简单的模型类似于只购买一张彩票，这几乎没有机会实现令人满意的性能。

（2）Metis 能解释全部的网络系统吗？诚然，Metis 不能解释所有基于深度学习的网络系统。例如，网络入侵检测系统（NIDS）用于通过正则表达式匹配数据包有效负载来检测恶意数据包[162]。之前的基于深度学习的方法引入了循环神经网络来提高 NIDS 的性能[143]。但是，由于循环神经网络基本上包含**隐式**的记忆单元，决策树无法仅使用**显式**决策变量来捕获策略。在未来，我们的目标是将 Metis 与循环单元结合起来，例如，使用循环决策树[163]。

（3）如何解释更深的深度神经网络？尽管我们的评估显示了三个基于深度学习的网络系统的令人满意的性能，但与其他领域（图5.2）中深度神经网络的应用相比，网络系统仍处于初级阶段：Pensieve 的隐藏层数甚至都不到 10 层。当前的方法是否能够扩展到具有更复杂神经网络的网络系统仍然未知。但是，一方面，Metis

可能是可扩展到更深的神经网络的，因为更深的神经网络（无论训练难度如何）有时具有与较浅的神经网络相同的表达能力[164-165]。另一方面，作为一个初步尝试，我们采用了传统的 CART 算法来训练决策树。在训练深度神经网络的过程中，可能会使用更优化的决策树表示[166]，以及树的正则化[167]，以更真实地解释策略。

（4）神经网络的泛化能力会受损吗？尽管神经网络的泛化能力仍在探索中，但毫无疑问，神经网络的泛化能力根源于大量的参数[168]。尽管 Metis 在本章的实验设置中表现良好（§5.5），但对于不同的数据集，解释的泛化能力仍需要进一步的研究。有两种方法可以进一步解决不同数据集上解释的泛化能力。一方面，研究人员可以分析解释的理论性能上限[169]。另一方面，网络运营商可以将解释结果部署到生产环境中，并评估在线性能。

（5）解释结果总是正确的吗？ Metis 旨在通过帮助网络运营商理解（并进一步排查）基于深度学习的网络系统来提供一种信心。但是，解释本身也可能会出错。事实上，研究人员最近发现了针对图像分类的解释系统的攻击[170-171]。尽管如此，我们的实验结果显示解释是经验上的健全的（§5.5）。由于解释是简洁且易于理解的，因此人类运营商可以轻松地发现罕见的错误解释。

5.7　本章小结

本章提出了 Metis，一种新的用于解释多种基于深度学习的网络系统的框架。Metis 将复杂的决策系统分类，并根据模型和分析它们的共同点提供相应的解决方案。我们将 Metis 应用于一个多媒体传输系统中。评估结果表明，Metis 基于系统可以以高质量解释基于深度学习的网络系统的行为。进一步的用例表明，Metis 可以帮助网络运营商设计、调试、部署和临时调整基于深度学习的网络系统。

第 6 章　数据通路应用层：自适应帧率缓解解码拥塞延迟

6.1　本章引言

像 5G 这样的新兴网络技术已经激发了学术界和工业界对高质量实时通信（RTC）应用程序的兴趣，聚焦于超高清（UHD），高帧率（HFR）和减少延迟等方面。具体的应用包括云游戏[20,172]，虚拟现实[173-175]和 4K 视频会议[176-177]等。一些高质量的实时音视频服务已经投入生产（譬如来自 Google 的云游戏[178]，来自 Microsoft 的云游戏[179]，来自 Nvidia 的云游戏[180]），例如，云游戏市场在 2020 年达到了十亿美元，预计增长率为 30%[181]。

为了实现这些应用程序，网络需要提供高分辨率、高帧率和低延迟的流媒体服务（§6.2）。比如，云游戏服务需要提供 1080p 以上的分辨率[178]，60fps 的帧率[182]，并且要求尾部的端到端延迟低于 100ms[7]。这种流媒体服务显着提高了用户体验，并为新的应用程序提供了可能。

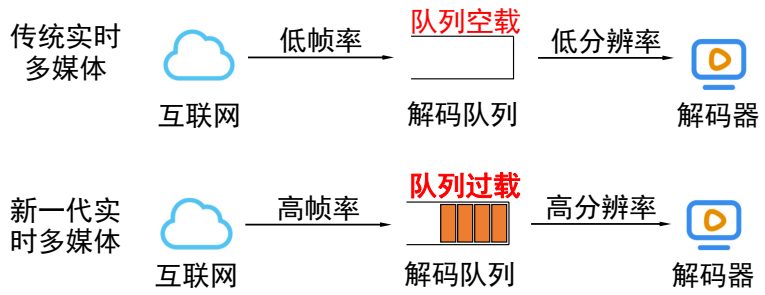


图 6.1　传统和高质量实时音视频应用程序之间的解码器队列比较

本文提出，为了实现高质量实时音视频服务，除了调整比特率以匹配网络容量之外，还必须调节解码器队列的排队。对于传统的标准质量实时音视频，解码一帧的时间远远小于帧之间的到达间隔时间。因此，解码器队列不是瓶颈，传统的实时音视频服务只需要调整比特率以匹配网络带宽。但是，在高质量实时音视频中，高帧率减少了客户端之间帧到达的间隔时间，而高分辨率增加了每帧的解码延迟。在解码器队列中，帧到达速率经常超过出队速率，导致队列很长。如图6.1所示，由于高帧率和高分辨率，当网络条件或解码器能力波动时，高质量实时音视频应用程序可能会导致解码器队列过载，从而导致尾部的高延迟。视频传输不仅需要调整比特率以匹配网络带宽，而且还需要与解码器队列容量协调。根据本文对腾讯 START 云游戏服务的测量[100]，不协调队列容量的视频传输可能会在客户端解码器队列中引入非常显著的排队延迟。此外，这种排队延迟占不满足高质量

实时音视频的延迟要求的延迟帧的很大比例，尤其是当网络延迟已经随着近期的基础设施发展而减少（例如边缘计算[90]）。根据我们的测量，所有总往返延迟大于 100ms 的帧中，57% 的帧在解码器队列中延迟超过 50ms（§6.3.1）。我们的调查发现，UHD 和 HFR 视频的未来需求将进一步加剧问题，即使随着解码硬件的发展（§6.3.1）。因此，对于高质量实时音视频而言，为了降低端到端延迟，降低解码器排队延迟是至关重要的。

然而，并不是所有的干预都能有效地调节解码器队列的排队（§6.3.2）。例如，解码延迟不受比特率的影响。它受分辨率的影响，但是调整分辨率需要客户端请求新的关键帧。这会消耗带宽，并且会导致几个额外的帧间隔的延迟。在客户端丢弃帧也需要新的关键帧，这也会产生相同的成本。因此，我们引入了一个自适应帧率（AFR）控制器，它控制编码器的帧率。降低帧率可以使解码器有更多的时间来处理帧。因此，它对减少解码器队列长度是有效的。此外，边缘流媒体服务提供短的 RTT，这意味着调整编码器帧率的控制循环是短的。

注意到，有一些现有工作也尝试调整帧率（例如 CU-SeeMe[183] 几十年前）。然而，解码硬件的发展使得这些工作在最近的十年中变得多余，而传统的实时音视频服务在最近的十年中主要受网络的限制。在本文中，我们展示了高质量实时音视频服务（如 UHD 分辨率，HFR 和严格的延迟要求）如何改变这一点。我们还以两种方式改进了这些提案。首先，现有的控制机制基于延迟或队列长度[33,184-185]，这些机制需要等待队列积累才能反应，因此反应速度很慢。AFR 不仅依赖队列长度，还依赖到达和服务过程来调整帧率。其次，不是所有增加的解码器队列延迟都需要降低帧率。例如，当队列延迟增加时，由于到达包的瞬时突发。因此，AFR 使用两个控制循环，它们在不同的时间尺度上调整帧率。

我们在两个模拟器和腾讯 START 的云游戏服务上实现了 AFR 控制器[100]。基于真实数据集的仿真和在线上真实业务部署证明了 AFR 可以有效地将尾部排队延迟减少 7.4×，从而减少由总延迟测量的帧卡顿率减少多达 2.2×（§6.6.1和 §6.6.5），并且开销很小。AFR 自 2021 年 2 月以来已经部署在腾讯 START 云游戏上服务了数百万个会话。收集的数据集和模拟代码也已经发布。

本章的主要贡献如下：

- 我们通过一个月的测量活动来证明控制解码器队列的排队延迟的重要性，并且发现了高质量实时音视频在严格延迟需求下的独特挑战 （§6.3）。
- 我们设计了一个层次化的帧率控制器，AFR，来控制解码器队列向着超短延迟的方向在不同的场景下进行高质量实时音视频 （§6.4）。
- 我们通过真实数据集的仿真和大规模线上业务部署评估了 AFR （§6.5）。我

们的评估表明，队列延迟和总端到端延迟都可以显着改善 （§6.6）。

6.2　背景：高质量实时音视频传输

高质量实时音视频应用正在从工业界和学术界吸引关注。最近，一系列高质量实时音视频产品已经发布，包括云游戏[178-180]，虚拟现实（VR）[186-188]和 4K 视频会议[189]。例如，通过生成高质量内容并通过互联网流式传输，用户可以通过低成本设备享受更好的视频质量。具体来说，高质量实时音视频具有以下特征，与传统的实时音视频应用不同：

- **高帧率**。传统的实时音视频通常以低帧率（LFR）24fps[190]提供内容。然而，高质量实时音视频需要高达 60fps 的帧率，其中一些甚至需要 240fps 的帧率[191]。
- **高分辨率**。大多数现有的实时音视频应用程序默认以 SD 分辨率提供（例如，Google Meet[192]的 360p）。相比之下，高质量实时音视频应用程序需要 1080p 到 4K 或更高的分辨率[193]。
- **严格的延迟要求**。此外，高质量实时音视频应用程序还具有严格的延迟要求。例如，视频会议需要 150ms 的往返交互延迟[190]，而游戏需要 100ms[7]。

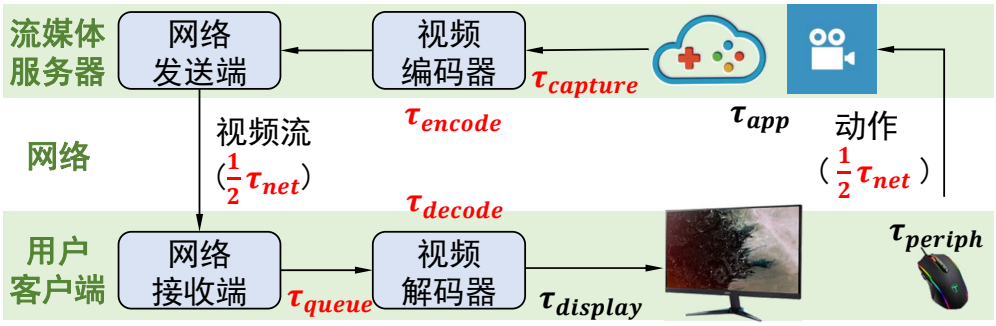


图 6.2　典型的实时音视频服务的分发流程

现有分发流程。为了更好地理解高质量实时多媒体传输的瓶颈，我们在图 6.2中介绍了现有实时音视频分发流程的关键组件。我们在红色中突出了基于我们的测量结果的高质量实时音视频服务尾端到端延迟的主要贡献组件。首先，视频编码器从视频源（例如，游戏应用程序[90,194]）捕获内容并将其编码为视频帧。然后，编码帧从流服务器通过网络发送到用户客户端。然后，在客户端，当从网络接收到新帧时，解码器将解码这些帧。最后，解码的视频帧将显示在用户的显示器上。

优化目标：低尾延迟。随着每个社区的智慧，每个组件的延迟都已经在最近的研究工作中进行了密集的优化。为了减少网络延迟，现有的提供商可以在边缘部

署流服务器[90,195]，引入低延迟拥塞控制器[49,60]，或建议用户使用有线连接。例如，最近的测量显示，云游戏服务可以在平均往返网络延迟为 20ms 的情况下提供实时音视频流[90,196]。同样，流编码器针对低延迟进行了优化，以满足高质量实时音视频服务中严格的延迟要求[33,197-198]。

同时，优化**尾**性能对于高质量实时音视频的用户体验也至关重要[5]。增加尾延迟会导致帧卡顿或冻结，降低用户体验。视频流媒体的体验质量评估框架通常单独计算卡顿时间作为用户体验的惩罚[40,199]。考虑到高质量实时音视频的高帧率，还需要关注第 99 或第 99.9 百分位数的更长尾部。例如，在 60fps 的帧率下，即使第 99.9 百分位延迟也会每 16 秒发生一次。特别是对于云游戏等应用程序，这样的延迟可能会导致游戏丢失（例如，在射击游戏中，当玩家被对手发现时会停顿）[7,200]。因此，控制尾延迟并减少帧卡顿对于高质量实时音视频至关重要。

约束 1：可靠传输。同时，交互式流媒体还需要可靠地传输每一帧。对于商业视频编解码器，丢失一部分帧将导致严重的图像质量下降。此外，丢失一帧也会导致后续帧模糊，因为帧之间存在依赖关系①。因此，现有的交互式流媒体服务通常尽其所能可靠地传输帧。例如，工业级框架（例如，WebRTC）[51,64]和学术工作[62-63,66]建议使用前向纠错（FEC）在接收器上尝试恢复丢失的数据包，如果失败，则会重新传输丢失的数据包[41]。

约束 2：高带宽成本。带宽成本仍然是云游戏服务的最大运营成本之一[201]。此外，为了实现令人满意的用户体验，交互式视频流必须以高视频分辨率和帧率（例如，云游戏需要 60fps 和 >1080p）进行传输，这需要高的好吞吐量来支持。鉴于运营成本低和用户高质量的要求，我们需要控制数据包丢失恢复的带宽成本。

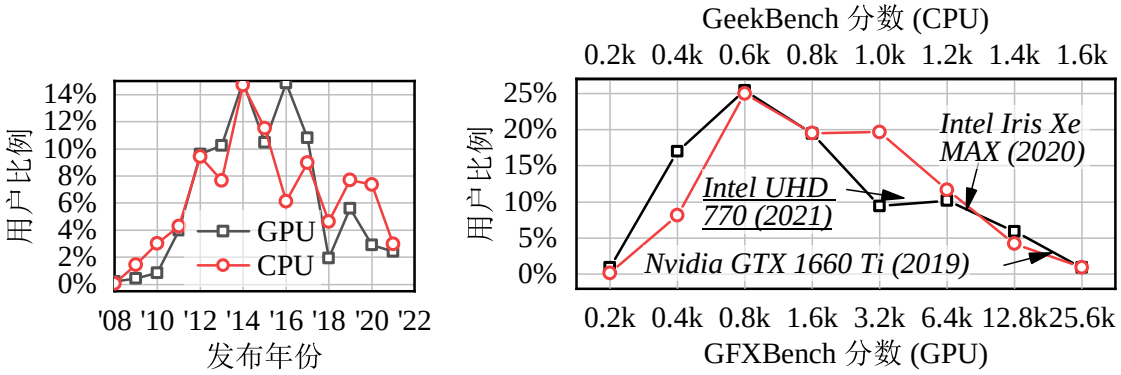
6.3　动机和挑战

在本节中，我们首先介绍了高质量实时音视频中队列恶化的形成原因（§6.3.1）。我们接着讨论了调整帧率的设计选择（§6.3.2）。最后，我们分析了有效地实现超短队列的独特挑战（§6.3.3）。

6.3.1　动机：糟糕的排队延迟

观察：解码器队列延迟是尾部延迟上升的关键贡献者。我们在图 6.2中对每个阶段的每个帧的延迟进行了分析。我们在 2021 年对腾讯 START 云游戏服务进行了一个月的测量，这一测量包含数万个用户，数千种不同的 CPU 和 GPU 型号。

① 例如，可扩展视频编码（SVC）允许有限的丢包，但会降低带宽效率并需要客户端支持[198]。

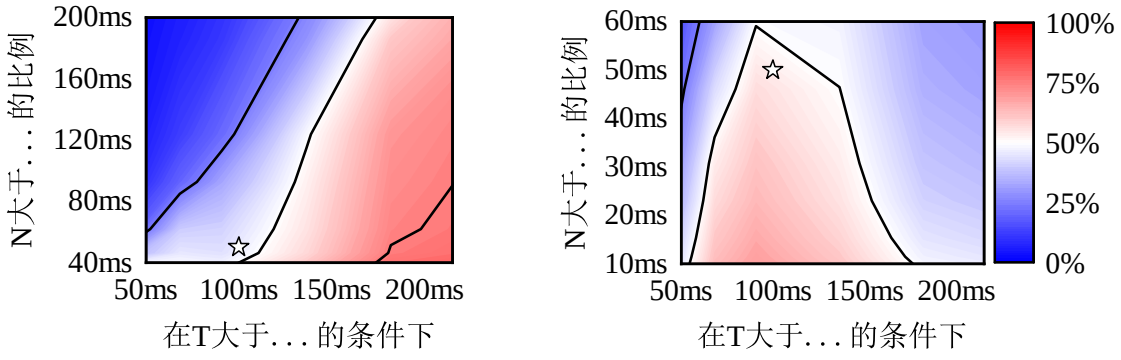


(a) 发布年份的分布　　(b) 基准测试得分的分布

图 6.3　用户设备在生产中的发布年份和基准分数分布

图 6.3展示了 CPU 和 GPU 的发布日期和基准分数。我们使用 GeekBench[202] 中的单核分数作为 CPU 基准分数，以及 GFXBench[203] 中的 Aztec Ruins Normal Tier 分数作为 GPU 基准分数。除非另有说明，本章中的所有测量都是从这个数据集中分析得到的。

根据测量结果，在流程中所有的组件中，网络、队列（在解码器队列中）和解码器延迟的 99 百分位数都大于 10ms。我们在图 6.2中将它们标记为红色。应用程序和编码器延迟的尾部很轻，因为它们在商业服务器上处理，这些服务器比网络和异构客户端稳定。因此，我们将在以下讨论中专注于网络、队列和解码器延迟。



(a) 网络延迟　　(b) 排队延迟

图 6.4　延迟组成部分的条件概率根因分析

我们调查这三个组件如何贡献于尾部总延迟的增加。对于每个帧，我们将 N、Q、D 和 T 分别表示为网络、排队、解码和总端到端延迟。然后，我们使用我们的测量值计算 $P(X > X_{th}|T > T_{th})$ 的条件概率，其中 $X \in \{Q, D, C\}$，X_{th} 和 T_{th} 是统计的阈值。高条件概率表明该组件更可能是 $T > T_{th}$ 的原因。我们使用不同的阈值计算条件概率，并在图 6.4中展示了网络延迟和排队延迟的结果。颜色表示 $P(X > X_{th}|T > T_{th})$，其中 $X \in \{N, Q\}$。星号表示 X_{th}=50ms，T_{th}=100ms。

如图所示，当分析传统实时音视频服务的根本原因时，尽管网络延迟通常应该在总延迟超过 200ms 时才被归因，但队列延迟在总延迟超过 100ms 的所有帧中占主导地位。我们的测量结果表明，在所有端到端总延迟超过 100ms 的帧中，排队延迟增加的情况更频繁：57% 的帧都有一个排队延迟超过 50ms（图 6.4中的星号）。考虑到高质量实时音视频的严格延迟要求 ~100ms，排队延迟的增加起着主导作用。

根因：超高清分辨率和高帧率共同导致排队延迟的增加。与低帧率流媒体相比，高帧率会通过减少帧之间的时间间隔来增加解码器队列的到达率。此外，与标清流媒体相比，超高清分辨率会通过增加每个帧的解码延迟来降低出发率。

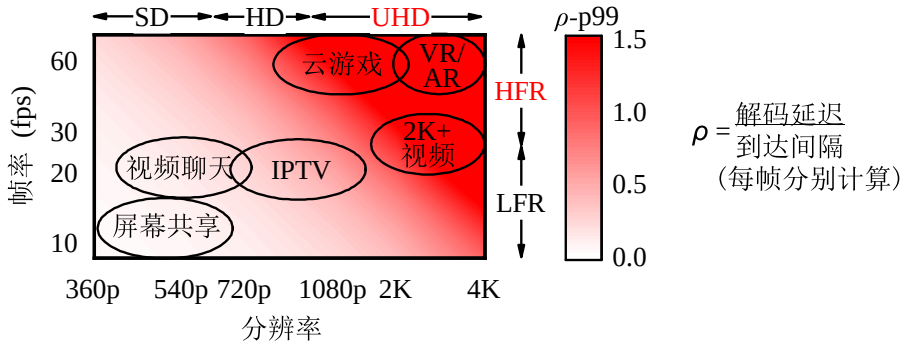


图 6.5　解码队列利用率 ρ99 分位示意图

特别地，我们通过展示 99 分位解码器队列利用率（图 6.5）来说明帧率和分辨率如何影响解码器队列的负载。我们将我们的测量结果的到达时间间隔和解码延迟分布缩放到其他帧率和分辨率。如图所示，对于传统的实时音视频服务（左下角），由于其低帧率和分辨率，解码器队列在尾部仍然具有 $\rho \ll 1$ 的利用率。但是，对于高质量的实时音视频应用（右上角），解码器队列在尾部（红色阴影）被大量过载，导致尾部的排队延迟增加。

问题来源于解码器性能的平均值和尾部值的不一致。事实上，我们测量的许多硬件解码器声称支持 UHD 和 HFR 视频（例如，Nvidia GTX 系列）。然而，根据我们的测量，支持 UHD 和 HFR 并不意味着**一致地**支持。例如，由于许多原因（包括客户端的过热[204]、CPU 调度 (§6.5.1) 和预测误差[205]），解码器的性能可能会波动，这对于应用程序来说很难控制。根据我们在生产设备上的测量，即使使用硬件加速，99 分位的解码延迟也为 18ms。请注意，60fps 的帧率下，帧之间的时间间隔为 16.7ms，因此 99 分位的解码延迟为 18ms，这意味着解码器的性能在尾部值上仍然不稳定。

我们进一步分析了其他组件和总延迟之间的必要性和充分性，并发现解码延迟的微小波动会导致排队延迟的增加。根据排队理论，当队列过载时，排队延迟会

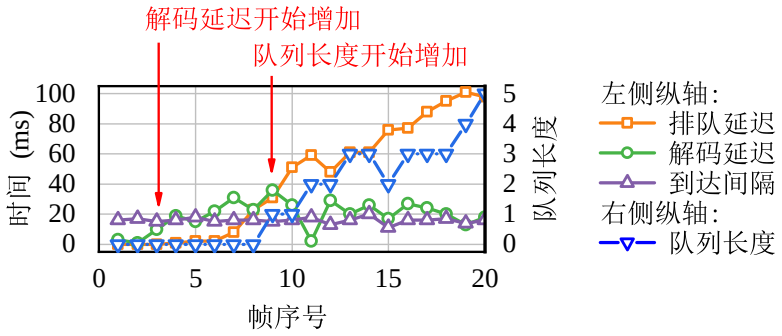


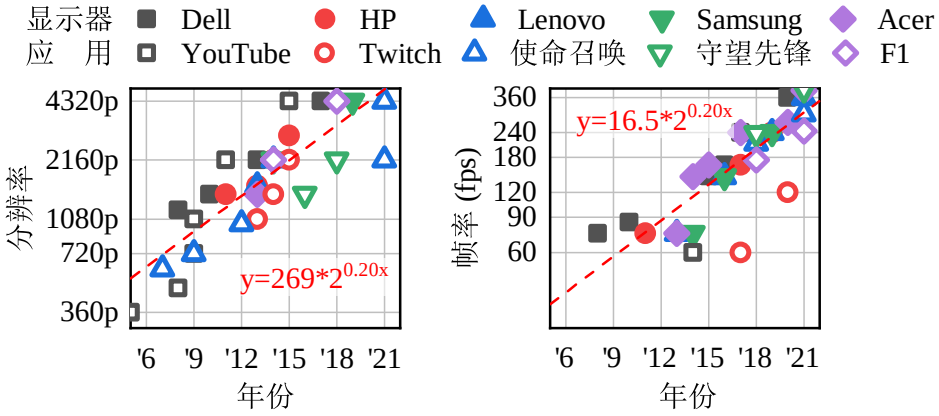
图 6.6　一个解码器队列堆积的说明性的例子

急剧增加[206]。这是因为当解码延迟不断波动时，排队延迟会累积所有前面帧的波动。特别是在高负载情况下，解码延迟的微小波动可能会导致排队延迟的数量级增加。我们建议读者参考[206]进行更多的理论分析。举例来说，我们在图 6.6中展示了我们生产服务中的一个跟踪。在跟踪中，帧间隔为 16ms，解码延迟为 18ms，而排队延迟平均为 54ms。解码延迟的连续增加，虽然在数量级上并不大（18ms），但是在时间上却是连续的。这种连续的增加导致了排队延迟的急剧增加。如果这个跟踪的概率为 1%，那么我们将得到 99 分位的解码延迟为 18ms，99 分位的排队延迟为 55ms。在这种情况下，尾部的排队延迟比解码延迟高得多，这也导致了超过一半的端到端卡顿，如 §6.3.1中分析的那样。

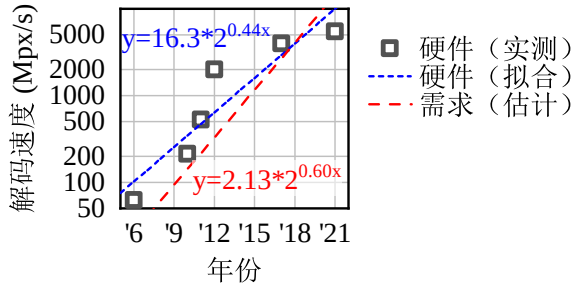
趋势：硬件解码器无法跟上视频高分辨率和高帧率的快速增长的需求。用户对视频的需求已经急剧增加，如图 6.7(a)所示。注意，需求的解码速度是帧率乘以分辨率的平方乘以宽高比。例如，YouTube 支持的最高分辨率和帧率从 2005 年的 360p@30fps（7Mpx/s）增加到 2015 年的 8K@60fps（2Gpx/s），平均每 14 个月增加一倍。新兴的服务如 16K[193,208]或 240fps[191]进一步表明了 UHD 和 HFR 流媒体的未来需求。

然而，解码硬件的解码速度并没有增长得足够快。我们总结了最近的学术论文中的视频解码器的解码速度[208-213]。如图 6.7(b)所示，解码硬件的解码速度每 27 个月增加一倍（蓝色虚线）。同时，我们还计算了视频需求的解码速度，每 20 个月增加一倍（红色虚线），并将其绘制在图 6.7(b)中。视频需求的解码速度增长得比解码硬件的解码速度（蓝色虚线）快得多，表明解码硬件无法满足 UHD 和 HFR 视频的需求。

除了使用最先进的硬件的用户，事实上现在的用户仍然有大量的低端和中端设备。这些用户设备，即使在同一代中，也可能非常多元化。例如，在图 6.3中，Intel Iris Xe 的性能是 Intel UHD 770 的 2 倍，即使后者的出厂时间事实上更新一些。因此，即使在同一代中，用户设备也存在异构性。此外，新的视频编解码器（例如，



(a) 前五大显示器厂商、两个平台 (YouTube 和 Twitch)、和三个游戏 (使命召唤、守望先锋和 F1) 的最大支持分辨率[207]



(b) 现有硬件的解码速度和应用需求的解码速度

图 6.7　解码硬件无法跟上视频高分辨率和高帧率的快速增长的需求

H.265）虽然具有更高的压缩比，但是甚至会使解码速度降低 60%[21,214-215]。在这种情况下，解码器和这些超高清的实时多媒体传输视频之间的不匹配将进一步加剧，使尾部的排队延迟成为一个持续的问题。

6.3.2　选择：控制正确的参数

在这里介绍本章调整帧率的动机。对于编码器，有三个参数可以独立设置，包括帧率、比特率和分辨率。编码器将根据当前内容自动优化其他参数（例如，量化参数）以实现目标帧率、比特率和分辨率。读者可以参考 [216] 了解有关视频编解码器的更多细节。

本节首先分析这些参数如何影响不同组件的延迟。当比特率增加时，因为网络拥塞可能会因之发生，网络延迟会增加。当分辨率增加时，由于解码器需要解码具有更大像素的帧，因此需要更长的时间来解码。排队延迟取决于入队速率（即帧率）和出队速率（即解码延迟）。相反，例如，如果比特率减少，但分辨率保持不变，则每个帧的解码延迟几乎不会减少，因为解码器的硬件设计。因此，发送端单纯依赖总延迟（例如，Salsify[33]）会导致对根因的分析出现歧义，无法采取有效的措施来减少延迟。

因此，我们需要独立控制各个参数以减少不同的延迟。为此，我们调整帧率来控制高质量实时音视频的排队延迟。当解码器和网络的波动导致排队延迟增加时，调整编码参数以减少排队延迟是至关重要的。在这种情况下，服务器上的编码器可以根据从客户端和网络收集的测量值相应地调整后续帧的帧率。我们可以动态指定编码新帧的时间戳。

总之，调整分辨率或丢弃帧是不切实际的，因为这会带来显著的带宽开销。静态地根据客户端模型选择帧率也不足以满足运行时的解码延迟的波动。此外，由于应用程序对用户系统的控制有限，因此对用户系统（例如，将应用程序固定到 CPU 核心）进行控制对于大规模的生产级服务也是不切实际的[217]。在帧率调整方面，需要注意的是，以前有过调整帧率的工作（例如，在几十年前就有了 CU-SeeMe[183]）。但是，正如我们在 §6.3.1中讨论的那样，随着分辨率和帧率的增加以及延迟要求的严格性，我们现在需要重新强调调整帧率的重要性。我们还表明，现在控制帧率的这种方法在互联网上是能够足够及时来对帧率进行控制的。

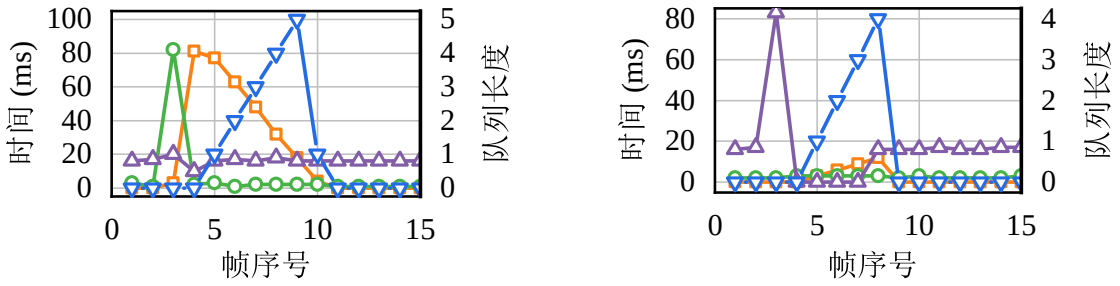
6.3.3　挑战

6.3.3.1　实现超短队列

为了实现超短的排队延迟，挑选合适的信号来通知控制器何时需要采取行动是具有挑战性的。现有的信号（队列长度[184]或排队延迟[33,185]）无法实现超短的排队延迟。由于到达或离开过程的波动，队列长度和排队延迟都只能在队列已经建立时才能观察到。对于图 6.6中的例子，虽然在第 3 帧开始时解码延迟开始增加，但只有在第 9 帧时才能观察到非零的队列长度。本章还在 §6.5.2中评估了基于队列长度和排队延迟的基线。

所以，这里需要捕捉**最早的信号**来感知潜在的排队延迟。因此，我们不是测量排队延迟，而是预测潜在的排队延迟增加。例如，受到最近在拥塞控制方面的进展的启发[78,218]，一种直接的方法是测量解码器队列的出队速率来估计潜在的排队延迟增加。

然而，当分析尾部的时候，到达过程也是波动的，这也可能导致排队延迟的增加。例如，网络延迟可能在第 99 个百分位数比中位数增加 10 倍[60]。为此，为了避免队列积累，我们扩展了[78,218]的设计：AFR 全面测量到达和离开过程，并根据排队理论控制排队延迟。我们在 §6.4.2中介绍了设计，并在 §6.5.2中评估了测量到达过程的必要性。



(a) 解码服务卡顿　　(b) 突发网络到达

图 6.8　两个来自在线跟踪的解码器队列的波动的例子

算法 6.1　层次化的自适应帧率控制.

输入: 入队过程 $\{A_n\}$, 出队过程 $\{S_n\}$, 队列状态 Q.
　　(A_n 表示到达时间间隔, S_n 表示帧 $\{n\}$ 的解码时间.)
输出: 目标帧率 f.
$f_0 = \text{StationaryController}(\{A_n\}, \{S_n\})$
$\alpha = \text{TransientController}(Q)$
$f = \alpha f_1$

6.3.3.2　处理各种事件

此外，高质量实时音视频中的解码器队列的形成原因也很复杂。正如我们在§6.3.1中介绍的，解码能力的静态退化可能导致解码器队列的积累，例如图 6.6中的跟踪。此外，解码器队列也可能由于瞬态的不利因素而积累。例如，从我们在生产中的经验来看，解码器可能会突然地出现 100 毫秒的解码延迟（例如图 6.8(a)中的第 3 帧，图例与图6.6相同）。无线信道中的突发干扰也可能导致几帧的突发到达（例如图 6.8(b)中的第 4 帧到第 8 帧）。在这两种情况下，解码器队列都会积累。由于这些瞬态波动是突然发生的，因此控制器很难通过测量入队和出队速率来做出反应。

因此，AFR 区分了队列积累的原因，并根据不同的时间尺度对波动做出反应。我们在 §6.4.2中设计了一个静态控制器来避免在高负载下的队列积累，以及在§6.4.3中设计了一个瞬态控制器来减少在突发情况下的排队延迟。

6.4　自适应帧率设计

本节首先分析了 AFR 的整体工作流程（§6.4.1），然后介绍了 AFR 的两个控制器（§6.4.2，§6.4.3）。

6.4.1　工作流程概述

AFR 整体的工作流程如算法6.1所示。具体来说，稳态控制器（§6.4.2）根据入队和出队过程的动态特性来维持队列在一个超短的目标值附近。通过测量这两个过程的统计量，AFR 根据排队论计算排队延迟的期望值。因此，帧率可以优化到给定的排队延迟目标（第 1 行）。瞬态控制器（§6.4.3）观察队列状态 Q（队列长度和排队延迟）并计算折扣因子 $\alpha \leqslant 1$（第 2 行），以进一步降低帧率，当队列形成时。最终的帧率是静态帧率 f_0 折扣后的值（第 3 行）。在这种情况下，AFR 可以对队列积累的各种情况做出反应。

6.4.2　稳态控制器

如上所述，我们测量入队和出队过程的统计量，并根据排队理论计算排队延迟的期望值。在这里，我们使用 Kingman 公式作为 G/G/1 队列的排队延迟的近似期望值。Kingman 公式是一种广泛使用的排队延迟近似公式[219]。与其他近似方法相比，本文中我们采用 Kingman 公式来估计排队延迟，因为它的估计来自于**入队和出队过程**，而不依赖于队列状态，这是队列延迟的最早信号。根据 Kingman 公式，排队延迟的期望值 τ_{queue} 如下：

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

其中

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

(μ_a, σ_a) 和 (μ_s, σ_s) 是到达和服务过程的均值和标准差，ρ 是队列的利用率。

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

由公式 6.1，排队延迟和下列因素相关：

- 队列利用率 ρ。当队列过载时（$\rho \to 1$），排队延迟也会增加。当前帧率和解码延迟决定了队列的利用率。
- 到达和服务的波动 c_a 和 c_s。当到达或服务过程波动时，排队延迟也会增加。
- 服务时间 μ_s。最后，排队延迟与平均解码延迟成比例。

因此，我们可以通过控制公式6.1的右侧来控制排队延迟。我们将排队延迟的期望值 $\mathbb{E}\{\tau_{queue}\}$ 设置为预定义的排队延迟目标 W_0。因此，目标帧率 f_0 可以通过公式6.4计算：

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

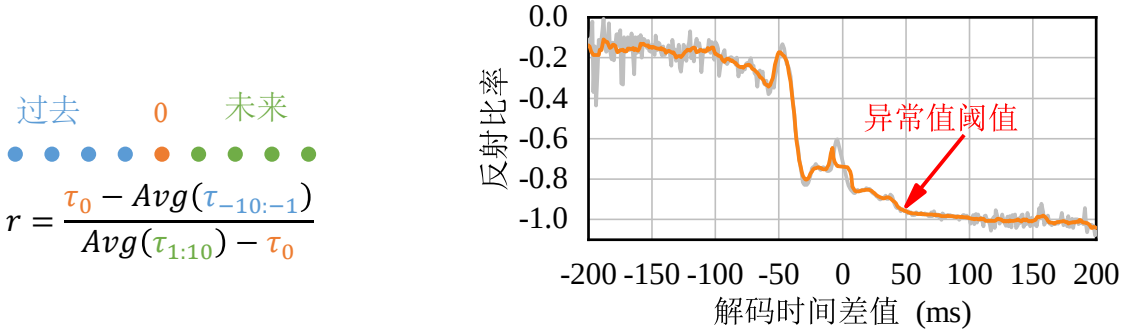
讨论：近似方法。 AFR 机制从原理上支持任何近似公式。同时，当然也还有其他研究工作来控制队列长度。例如，最近的拥塞控制研究[78,218]直接设置目标利用率（例如，设置 $\rho = 0.95$）并计算入队率。如 §6.3.3中讨论的那样，本章采用 Kingman 公式来分析 textsf 到达和服务过程。本章还在 §6.6.1中评估其他基线的性能。

队列动态性的测量。根据公式 6.4，我们需要测量到达和服务过程的均值和方差。类似于 TCP 中的 RTT 测量[220]，我们采用指数加权移动平均（EWMA）和指数加权移动方差（EWMV）来估计公式 6.1和6.2中的 $\mu_s, \sigma_s, \mu_a, \sigma_a$。

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

其中，x_n 是到达时间 A_n 或服务时间 S_n。$\hat{\mu}_n$ 和 $\hat{\sigma}_n$ 是 EWMA 和 EWMV。ξ_μ 和 ξ_σ 是均值和标准差的折扣因子，用于权衡精度和灵敏度。

然而，由于到达和服务过程的突发性（§6.4.1），到达和服务过程的值可能会发生显著偏离。例如，图 6.8(a)中的第 3 帧解码时间为 82ms，而其他帧都低于 4ms。这些异常值会显著偏离长期的稳态统计估计。事实上，正如本章在 §6.4.1中讨论的那样，这些偶然事件是由瞬态控制器来处理的。因此，AFR 需要过滤掉这些异常值，以精确地估计到达和服务过程的稳态状态。由于解码延迟的分布高度偏斜，基于标准差的现有异常值移除机制（例如，3σ 规则[221-222]）会区分稳态状态转换和异常值。



(a) 示意图　　(b) 生产环境下的测量结果

图 6.9　通过反射移除异常值

为了捕捉状态转换的过渡，同时消除偶然的异常值的影响，我们引入了一种基于生产中的先验知识的异常值移除机制。这里的关键发现是**解码延迟差**（$S_n - S_{n-1}$）与异常值的概率相关。例如，20ms 的解码延迟增加可能是稳态状态转换（图 6.6）。然而，80ms 的突然增加可能是异常值，这通常是图 6.8(a)中的瞬态控制器要处理场景。这是因为商业解码器通常能够在平均帧率 24fps 的情况下解码帧。根据我们

的测量，当解码延迟差大于 50ms 时，该帧是异常值的概率为 95%。因此，稳态控制器在计算中移除解码延迟差大于 50ms 的帧。这些帧的控制权交给瞬态控制器。

此处进一步基于生产中的测量来验证这一直觉。如图 6.9(a)所示，我们用**反射比**（r）来量化异常值，它展示了解码延迟的恢复情况。分子是当前解码延迟（τ_0）与之前 10 帧的平均解码延迟（$\tau_{-10:-1}$）之间的差值，分母是 τ_0 和未来解码延迟之间的差值。对于依赖于瞬态控制器的异常值（例如，图 6.8(a)中的第 3 帧），它们的反射比接近-1。这是因为之前的帧和之后的帧具有相似的解码延迟，而异常值具有更高的解码延迟（$\tau_0 \gg \tau_{-10:-1} \approx \tau_{1:10}$）。

我们接下来画出了解码延迟差（$\tau_0 - \tau_{-1}$）与所有具有相同解码延迟差的帧的平均反射比（r）之间的关系，如图 6.9(b)所示。测量细节见 §6.5.2。当解码延迟差大于 50ms（用红色箭头标记）时，平均反射比小于-0.95，表明该场景中的大多数帧是异常值。因此，AFR 中的稳态控制器不会计算解码延迟差大于 50ms 的帧。

收敛时间分析。为了帮助运营商更好地理解稳态控制器的行为，我们研究了服务过程的状态转换期间稳态控制器的收敛情况。此处想回答以下问题：在从稳态状态 (μ_1, σ_1) 转换到 (μ_2, σ_2) 的过程中，稳态控制器需要多长时间才能收敛到新的帧率，并消除由于转换而积累的队列？

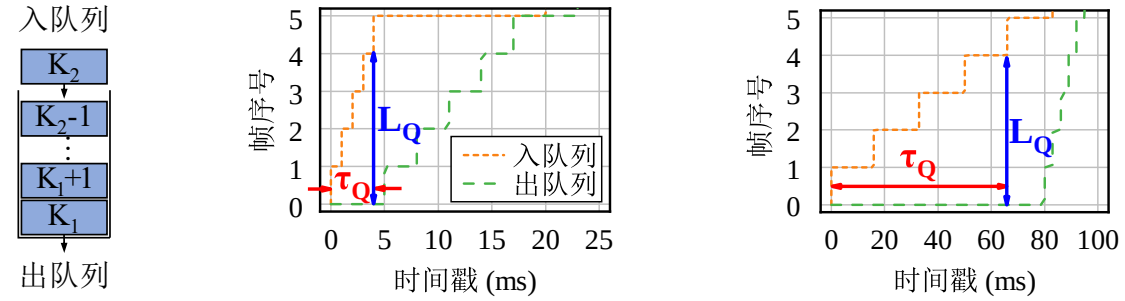
我们在这里勾画了主要的结论。当 AFR 的控制回路（往返延迟）为 τ 帧时，稳态控制器的收敛时间 T_0 与 τ 和 W_0 成正比，并且对于大多数场景是可接受的。例如，当 AFR 的平均控制回路为一个帧的到达间隔（τ=1），并且 W_0=2ms 时，稳态控制器可以在 2 帧内收敛到新的稳态状态。

6.4.3　瞬态控制器

瞬态控制器是为了处理突发的队列积压而设计的（§6.4.1）。因此，我们首先需要理解如何应对这些队列积压。

6.4.3.1　理解队列积压

如图6.8(a)和6.8(b)所示，解码器服务的停滞和网络到达的突发都会导致队列长度突然增加。我们在图 6.10中说明了两种情况下的入队和出队事件。Y 轴表示累积的入队/出队帧。例如，图 6.10(b)中的入队曲线从 1 增加到 2，表示第 2 帧在 1ms 时入队。在图 6.10(b)中，有 5 帧在 4ms 内到达客户端，当第 5 帧到达并观察时，队列长度为 4（蓝色箭头所示）。在图 6.10(c)中，解码器需要 80ms 才能解码第 0 帧，因此队列中的帧无法出队到解码器。因此，当第 5 帧到达时，它也会观察到队列长度为 4。



(a) 队列.　(b) 突发网络到达.　(c) 解码服务卡顿.

图 6.10　突发网络到达和解码器解码卡顿的区别

然而，突发网络到达和解码器服务的停滞应该分开处理。在突发网络到达的场景中，总延迟的瓶颈仍然在网络中，因为它的网络延迟很长。只要解码器是正常的，即使多个帧同时到达队列，它们也可以高效地处理（图 6.8(b)）。因此，队列将在短时间内排空，我们不需要降低帧率。相反，解码器服务的停滞会大大增加后续帧的排队延迟，并需要适应（图 6.8(a)）。因此，我们需要区分这两种情况。

因为两种情况都会导致队列长度增加，因此不能仅通过队列长度来有效区分它们。我们的观察是，我们可以通过队列中第一帧的停留时间来区分它们。如图 6.10(a)所示，在第 2 帧到达时，第 1 帧的停留时间 τ_Q 和队列长度 L_Q 由第 2 帧观察到如下：

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

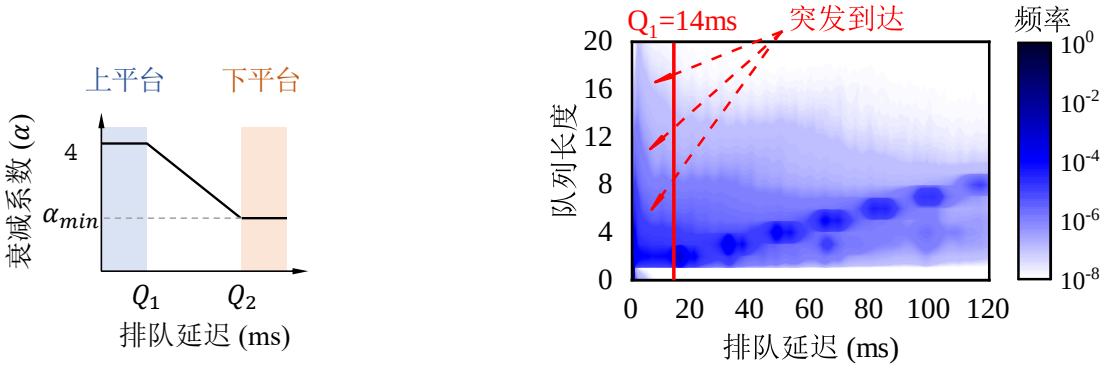
其中，$t_{enq}^{(i)}$ 是第 i 帧的入队时间戳，第 K_1 帧是队列头部的帧。对于突发网络到达，由于帧同时到达解码器队列，因此当最后一帧到达时，第一帧只排了很短的时间的队。例如，图 6.10(b)中的 τ_Q 为 4ms（标记为红色）。相反，对于解码器服务的停滞，队列头部的帧已被阻塞了很长时间，导致图 6.10(c)中的高 τ_Q 为 66ms。因此，我们使用 τ_Q 来调整瞬态控制器中的帧率。

6.4.3.2　反馈控制

瞬态控制器的设计空间是找到一个将衰减系数 α 和排队延迟 τ_Q 联系起来的映射。由于瞬态控制器是为了根据稳态控制器的结果来降低帧率，因此 α 的可能范围满足：

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

其中，f_{min} 和 f_{max} 是应用程序所需的最低和最高帧率。由于较长的 τ_Q 表示队列的负载更严重，因此衰减系数应随着 τ_Q 的增加而减小。此外，α-τ_Q 映射还应具有



(a) α-τ_Q 映射　　(b) L_Q-τ_Q 的出现频率

图 6.11　瞬态控制器的示意图与测量情况

以下属性：

第一，避免过度反应。如前所述，对于突发网络到达，由于大量到达的帧，τ_Q 也会略微增加。但是，由于这种瞬态队列积累会很快被清除。因此，只要解码器是正常的（图 6.10(b)），我们就不应该降低帧率，我们需要引入一个上平台（如图 6.11(a)）来避免过度反应。在上平台中，当观察到非零但较小的 τ_Q（$0 \leqslant \tau_Q \leqslant Q_1$）时，瞬态控制器不会降低帧率。上平台的阈值 Q_1 应基于测量结果来设置。我们从帧中测量 L_Q 和 τ_Q 的结果，并在图 6.11(b)中呈现。图 6.11(b)中一系列线性分布的深蓝色点表示 L_Q 和 τ_Q 是线性相关的。图 6.11(b)中靠左轴的峰值（用红色虚线标记）表示具有较长 L_Q 但较短 τ_Q 的帧。这些帧是由于突发网络到达而导致的瞬态队列积累。因此，我们将 Q_1 设置为过滤掉这些瞬态到达相关的峰值（例如，Q_1=14ms 在我们的部署中，图 6.11(b)中的红线）。

第二，及时响应。由于高质量实时多媒体应用程序的严格延迟要求，队列延迟会大大降低用户体验。因此，我们需要控制图 6.11(a)中的映射斜率，以有效地减少队列延迟。由于 α 有下限，我们可以通过引入一个**下平台**来控制映射的斜率，如图 6.11(a)所示。我们将 Q_2 设置为最大可容忍的队列延迟：

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

其中，$\tau_{network}$ 是网络往返延迟，τ_{decode} 是解码延迟 μ_s。$Deadline$ 是应用程序的总延迟要求。根据用户在人机交互中的经验和我们的运营经验，我们将 $Deadline$ 设置为 100ms[7]。

6.5　系统实现

我们将 AFR 实现在一个基于逐帧的真实数据驱动的仿真器中，然后将 AFR 部署到一个生产环境中的高质量实时多媒体服务中。本节介绍了仿真器的设

计 (§6.5.1)，仿真设置 (§6.5.2) 和部署设置 (§6.5.3)。

6.5.1　仿真器设计

为了公平地比较和重放不同的队列控制算法的数据集，我们设计了一个简单的仿真环境，该环境模拟了实时音视频的动态。仿真器维护解码器队列，并从在线服务中重放数据集，其中数据集包含解码延迟、网络延迟、原始排队延迟以及每个帧的到达时间戳。具体来说，根据数据集中的时间戳，帧按顺序到达解码器队列，等待出队，然后根据跟踪中的解码延迟进行解码。为了避免频繁向服务器发送帧率调整请求，帧率被量化为 5fps 的级别，这也是我们在线部署的跟踪中的帧率。我们实现了 CPU 时间切片可能带来的潜在干扰：由于帧的获取对解码器依赖于 CPU，因此可能会出现将帧从队列获取到解码器需要等待几毫秒才能被 CPU 调度的情况[223]。因此，我们在仿真器中进一步对此类延迟进行了跟踪，并引入了调度等待时间。我们还根据我们在 §6.6.4中测量的响应时间来实现编码器的响应时间。

6.5.2　仿真设置

表 6.1　此处数据集按照不同客户类型的分布

	类别	会话	帧数量	游戏时长
(1)	Windows+ 有线网	2.97 万	63.5 亿	3.42 万小时（3.9 年）
(2)	Windows+WiFi	0.64 万	11.2 亿	0.62 万小时（8.6 月）
(3)	MacOS+ 有线网	0.04 万	0.41 亿	0.02 万小时（8.3 天）
(4)	MacOS+WiFi	0.21 万	2.16 亿	0.11 万小时（1.5 月）
	总计	3.81 万	77.3 亿	4.17 万小时（4.8 年）

数据。我们在两种客户端（Windows 和 MacOS）和两种接入网络（以太网和 WiFi）上测量了腾讯云游戏服务的帧级统计信息（在 §6.3.1中介绍过）。我们在 2020 年 12 月的 24 天中在一个生产服务器集群中的记录了每一个接收到的帧。这导致了一个有 77.3 亿帧和 4.17 万小时游戏时间的数据集（表 6.1），这是我们所知的最大的交互式流媒体帧级数据集。

参数设置。AFR 中有几个参数需要确定。除了与瞬态控制器（§6.4.3）相关的参数外，我们将瞬态控制器中的 W_0 设置为 2ms，EWMA 中的折扣因子 $\xi_{arrv} = 0.033$ 和 $\xi_{serv} = 0.25$。6.6.3节讨论了这些设置的敏感性以及它们对性能的影响。

指标。在评估中，我们主要测量延迟（包括排队延迟和端到端总延迟）。正如我

们在 §6.2中讨论的，交互式流媒体中的延迟与其他视频质量指标（例如 PSNR[31]或 SSIM[30]）是正交的。延迟是本文的主要优化目标。我们在 §6.6.4中证明了 AFR 对视频质量的损失是可以忽略不计的。

基线方案。为了评估 AFR 的性能，我们实现了现有的帧控制机制：

- `DropTail` 是 WebRTC[184]中的帧控制机制。当帧溢出队列时，客户端首先清除队列，然后请求新的关键帧，并最终丢弃所有帧，直到下一个关键帧到达。我们将队列容量设置为 16 帧。
- `QLen-S` 观察当前队列长度，如果队列长度 ⩾1，则在编码器之前跳过帧，并在队列长度 <1 时恢复。
- `QWait-S`。我们从现有的学术工作中迁移了帧控制机制到我们的仿真器[33,185]，并将信号从总延迟替换为排队延迟以更好地减少排队延迟。由于这些基线方案不是为 100ms 的严格延迟要求而设计的，因此我们还使用我们的跟踪来微调它们的参数。`QWait-S` 中如果排队延迟 ⩾32ms，则在编码器之前跳过这些帧，并在排队延迟 <4ms 时恢复。

此外，为了评估 AFR 中不同组件的有效性，我们还实现了 AFR 的不同变体：

- `AFR-QLen`. 我们使用基于当前队列长度的反馈算法来证明基于队列状态控制帧率是不足的：它观察到每个帧到达时的当前队列长度，并将队列长度 {0, 1+} 映射到帧率 {60, 24}fps。
- `AFR-QWait`。一个反馈算法将当前排队延迟 $\{(0,4),(4,8),(8,12),(12,\infty)\}$ms 映射到帧率 {60, 48, 36, 24}fps。这些参数也是通过我们的数据集进行微调的。
- `AFR-TX`。为了证明测量到达和服务过程都是有效的，我们进一步实现了一个基于出队速率的算法。`AFR-TX` 测量出队速率，并将目标帧率设置为 $\rho = 0.8$，其中 ρ 已经通过我们的跟踪进行了调整。出队速率是解码时间的倒数，即出队速率 =1/解码时间。
- `AFR-Kingman`。我们还单独评估了 AFR 的稳态控制器，以进一步说明瞬态控制器的有效性。
- `AFR`。最后，我们将本文中的所有优化（包括稳态和瞬态控制器）全部放在一起。

第6.6.3节讨论了这些设置的敏感性以及它们对性能的影响。

6.5.3　部署设置

我们最终将 AFR 部署到一个云游戏服务（腾讯 START 云游戏）中。该游戏服务使用 H.264 编解码器来增加硬件解码的覆盖率，并针对异构客户端进行优化 ①，并定制编解码器以优化游戏性能。该游戏目前支持 13 款生产级游戏，包括动作冒险、第一人称射击和实时策略游戏。为了优化网络延迟，该服务使用多接入边缘计算进行加速[90,195,225]：用户被分成几十个操作区域，每个区域的地理直径为数百公里。每个运营区域都部署了云游戏服务器集群，平均往返网络延迟为 15ms。

帧率适应的算法部署在了客户端上。AFR 控制器持续测量解码队列的统计信息，并在必要时向边缘服务器发送请求以调整帧率。边缘服务器将帧率调整请求转发给视频编码器和游戏应用程序。新帧将根据新的帧间间隔生成。6.6.4节评估了视频编码器和游戏应用程序的响应时效性和开销。

6.6　评价

我们在如下方面评估了 AFR：

- **延迟提升**。本小节展示了性能提升：AFR 的帧数比和总延迟比现有基线提高了 2.1×-26× 和 13%-2.2× (§6.6.1)。
- **保持帧率**。本小节之后证明 AFR 对帧率相关的指标的影响很小 (§6.6.2)。
- **参数敏感性**。本小节的评估表明 AFR 中的参数可以设置为广泛的范围，以获得与微调基线相比的性能提升 (§6.6.3)。
- **微基准**。本小节进一步证明了 AFR 中的帧率调整的及时性、开销和图像质量对于在线部署是令人满意的 (§6.6.4)。
- **实际部署**。本小节最终将 AFR 部署到一个生产云游戏服务中，并证明了它的可行性 (§6.6.5)。

6.6.1　延迟提升

我们对比了使用 AFR 和基线算法在四个数据集上的排队延迟和总延迟（表 6.1）。本小节测量排队延迟的两个维度：图 6.12展示了 99%ile 排队延迟和排队延迟大于 50ms 的帧数比。我们首先分析了 AFR 与三种现有机制（`DropTail`、`QLen-S` 和 `QWait-S`）对比的结果。AFR 可以将 99 百分位排队延迟减少 1.9× 到 7.4×，并且排队延迟大于 50ms 的帧数比减少 2.1× 到 26×。在这种情况下，99 百分位排队延迟可以压缩到 6.9ms。这表明 AFR 可以有效地实现超短的排队延迟。AFR

① 硬件解码具有比软件解码更短的解码延迟，并支持更高的帧率。H.264 具有比其他高级编解码器更高的硬件解码支持率[224]。

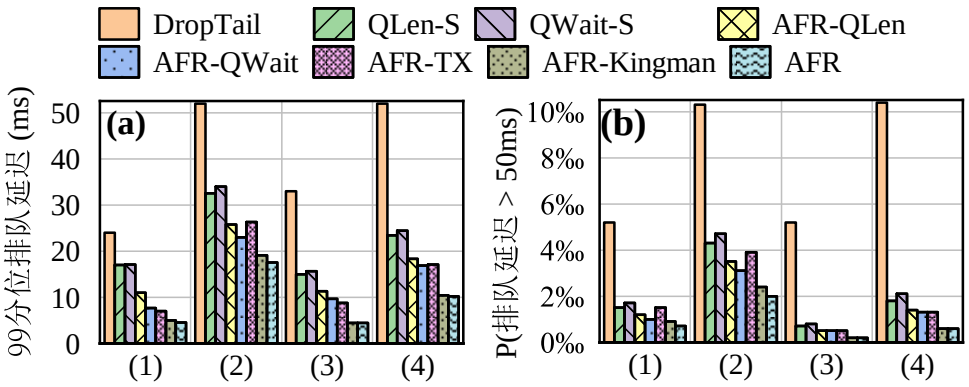


图 6.12　排队延迟的仿真结果

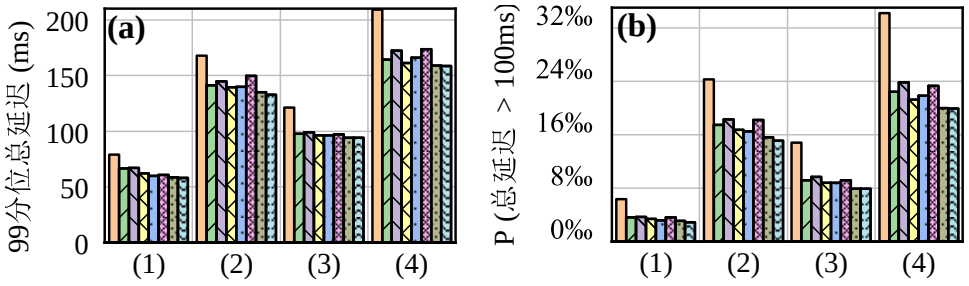
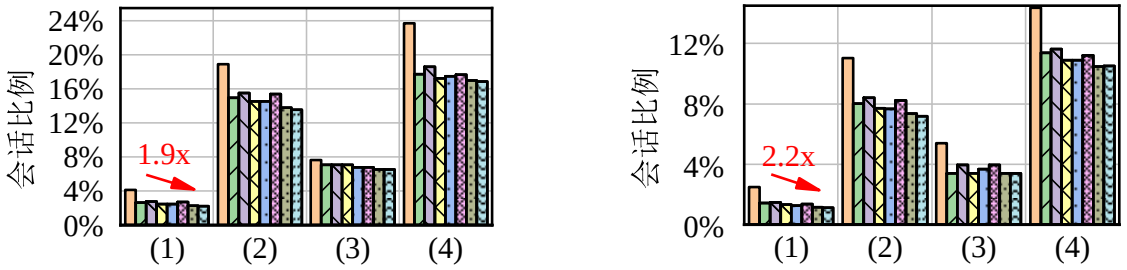


图 6.13　总延迟的仿真结果



(a) 卡顿率 >5% 的会话.　　(b) 卡顿率 >10% 的会话.

图 6.14　会话中不同卡顿帧的比例

还在**总端到端延迟**上表现出令人满意的性能提升，这直接与用户体验相关。AFR 将 99 百分位总延迟提高了 27% 到 36%，并且将总延迟大于 100ms 的帧数比提高了 1.6× 到 2.2×。我们还测量了**会话卡顿比**，即会话中总延迟大于 100ms 的帧数比。然后，我们测量了会话卡顿比大于 5% 和 10% 的会话比例，这表明有多少用户会遇到不满意的体验。图 6.14展示了结果。对于云游戏服务的主要人群（表 6.1中的类别 (1)），AFR 将卡顿会话减少了 17% 和 21%。对于其他类别，卡顿会话的比例也减少了 5% 到 37%。AFR 可以显着改善高质量实时流媒体服务的用户体验。

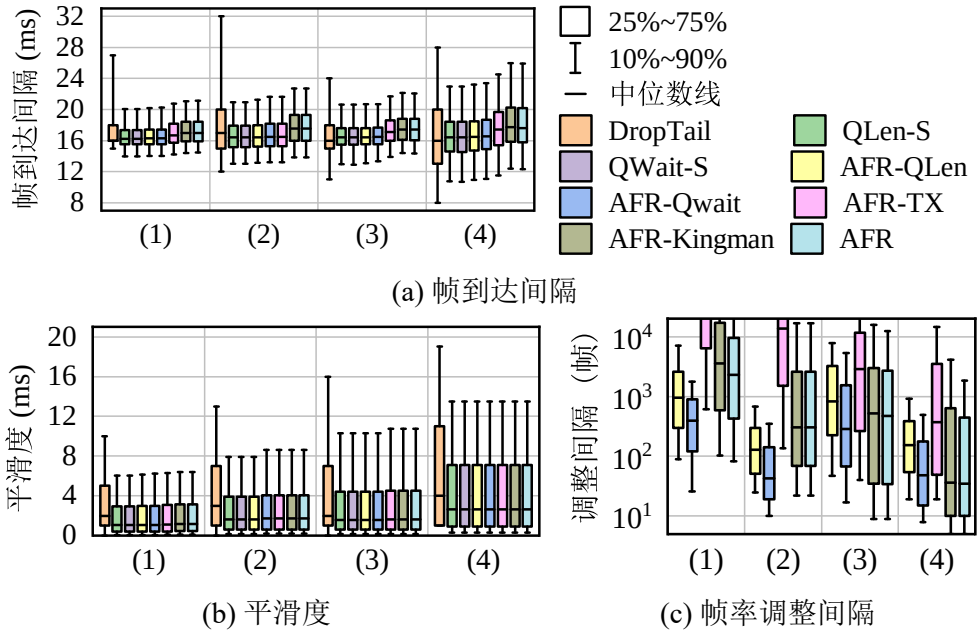
我们进一步通过对比 AFR 的不同变种来了解性能提升。与 `DropTail` 相比，基于队列状态的基线（`AFR-QLen`、`AFR-QWait`）可以有效地减少排队延迟，这表明主动控制排队延迟是必要的（§6.3.1）。与 `QLen-S` 和 `QWait-S` 相比，控制帧率比从编码器跳过帧更好。这是因为跳过帧会大大降低尾帧率，因此基线的参数是针对该帧率进行调整的（§6.6.3）。`AFR-TX` 比基于队列状态的基线（`AFR-QLen`、`AFR-QWait`）可以进一步减少排队延迟，这表明观察服务过程可以提前知道潜在的服务质量降级，并有效地采取行动，验证了我们在 §6.3.3中的分析。`AFR-Kingman` 进一步提高了性能，比 `AFR-TX` 提高了 10%。这说明高质量实时流媒体的不稳定到达也会影响解码器队列的估计。`AFR` 最终将尾部排队延迟再减少了 2-4%，表明瞬态控制器需要处理突发情况。

此外，我们还发现 AFR 在网络条件更好时性能更好。在两组以太网数据集（类别 (1) 和 (3) 的 55% 和 37%）上的性能提升比 WiFi 数据集（类别 (2) 和 (4) 的 35% 和 27%）更大。考虑到下一代接入网络的部署（例如 5G 和 WiFi 6），控制解码器队列的必要性将更加明显。

6.6.2　帧率保持

此外，我们还测量了 AFR 对帧率的影响。我们首先测量了每个帧到达客户端之间的帧间隔时间。例如，60fps 的帧率应该会有 16.7ms 的帧间隔时间。我们调整每种算法的参数，使其 99 百分位的帧间隔时间保持在相同的水平（详见 §6.6.3）。因此，对于 10-90 百分位，如图 6.15(a)所示，除了 `DropTail` 之外的大多数算法都是互相可比的。与已部署的 `DropTail` 机制相比，AFR 甚至可以提高尾用户感知的帧率，这是因为它更好地管理了帧丢弃的问题。AFR 将中位帧率降低了 3%-9%，这对用户带来了可忽略的体验质量（QoE）降低，尤其是当考虑到延迟的改进[12,226]。

我们进一步测量了帧率的平滑度，这可能也会对用户体验产生潜在影响[199]。我们将**帧间隔时间的差异**作为帧率平滑度的指标，并在图 6.15(b)中呈现结果。除了 `DropTail` 之外，所有基线和 AFR 的帧间隔时间差异都相似，并且比 `DropTail`



(a) 帧到达间隔

(b) 平滑度

(c) 帧率调整间隔

图 6.15　帧率保持结果

好。这主要是因为 `DropTail` 中的帧丢弃会导致突然增加的帧间隔时间差异。此外，我们还测量了帧调整间隔，并在图 6.15(c)中呈现了分布。AFR 的中位调整间隔是数百到数千帧，远远长于帧率调整的响应时间（§6.6.4）。

6.6.3　参数敏感性

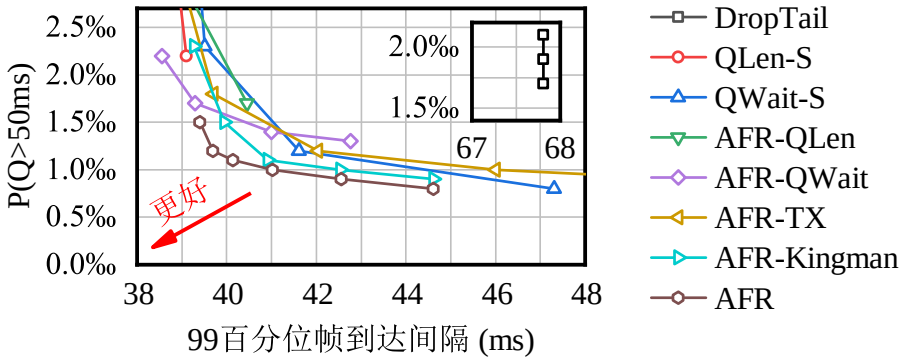


图 6.16　尾部到达间隔和排队延迟的权衡

我们接下来评估 AFR 和其他基线的参数敏感性。我们在 §6.5.2中调整了 AFR 和其他基线的参数：`QLen-S` 和 `QWait-S` 的跳过/降低帧率阈值，`AFR-QLen` 和 `AFR-QWait` 的映射，`AFR-TX` 的 ρ，以及 `AFR-Kingman` 和 AFR 的 W_0。我们在图 6.16中呈现了 Cat. (1) 跟踪的排队延迟 >50ms 的帧比例（P(Q>50ms)）和 99%ile 的到达间隔时间。左下角的算法表明它在排队延迟和帧率之间具有令人满意的权衡。

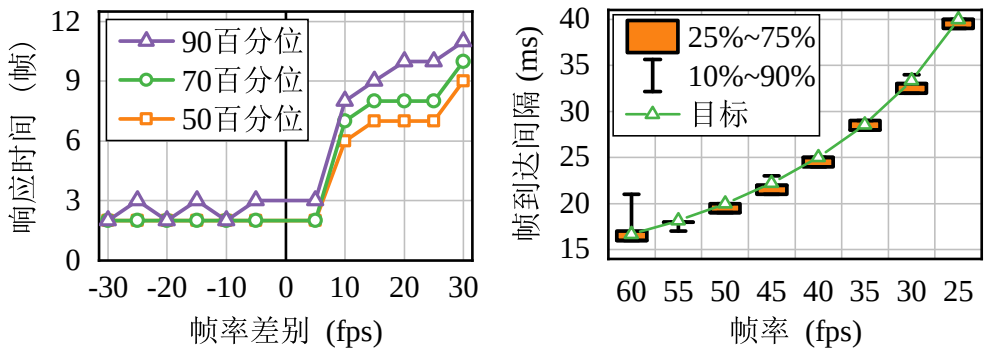
如图所示，AFR 在各种设置下均优于所有其他基线，实现了更好的排队延迟和

帧率之间的权衡。基于队列长度基线在实现超短排队延迟方面受到挑战：即使使用最极端的参数（只要队列长度不为零，就跳过/降低帧率），QLen-S 和 AFR-QLen 只能实现 0.22% 和 0.17% 的 P(Q>50ms) 卡顿率，远高于其他基线。这符合我们在§6.3.3中的分析，队列长度作为控制队列的信号太粗糙，无法实现超短目标。同时，基于跳帧的基线算法可以实现比基于帧率的算法更低的排队延迟，但是帧间隔时间更高。所有算法的参数都根据图 6.16进行调整，以便通过对齐 99 百分位的到达间隔时间来调整它们。

我们同时还评估了不同的排队延迟和总延迟的百分位数受 W_0 设置的影响。AFR 对 W_0 的设置反应敏感，表明运营商可以通过调整 W_0 来有效地平衡总延迟和帧率。我们还在 [227] 中评估了短期控制器 (§6.4.3) 中的指数加权移动平均（EWMA）和指数加权移动方差（EWMV）的衰减系数 ξ 的敏感性，展示了运营商应该如何设置这些参数来平衡精度和灵敏度。

6.6.4　微基准

我们同时评估了 AFR 在该云游戏的一个测试床上的一些基准性能。



(a) 调整的及时性.　　(b) 帧率的稳定性.

图 6.17　帧率调整的有效性

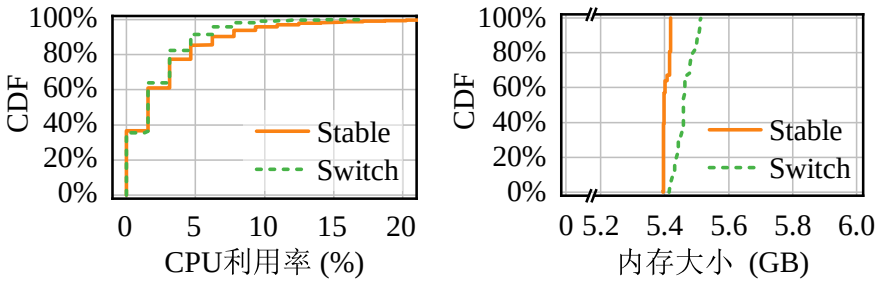
6.6.4.1　帧率调整的有效性

我们首先测量了视频编码器的响应性和精度。我们枚举了 {25, 30, ⋯, 60}fps 之间的所有帧率切换，并测量了编码器需要多少帧才能稳定地输出新帧率的视频流。以帧为单位测量的响应时间（即**响应帧**）在图 6.17(a)中呈现。对于每组设置，我们重复实验 100 次以消除随机性。当降低帧率时，90 百分位响应时长小于 3 帧，表明编码器和游戏应用程序可以及时降低帧率。这可以有效地减轻解码器队列的过载。当显着增加帧率时，帧率可能会稍有延迟。这是因为客户端的帧率遵循短板效应。编码器或游戏应用程序降低帧率都会导致客户端的帧率下降，而增加帧率需要两个组件都增加。即使如此，尾响应时长也小于 10 帧，远小于调整间隔

（图 6.15(c)）。

我们接下来测量了编码器的帧率稳定性。我们将帧率设置为上述几个级别，并测量每帧之间的到达间隔时间。对于每个帧率，我们测量 30,000 帧的到达间隔时间，并在图 6.17(b)中展示了结果的分布。每帧之间的到达间隔时间大多数都围绕目标帧率。因此，与视频流的比特率波动不同[22]，帧率可以由编码器精确控制。

6.6.4.2　帧率调整的开销



(a) CPU 利用率　　(b) 内存利用率

图 6.18　帧率调整开销

我们进一步测量了边缘服务器上帧率调整的潜在处理开销。为了放大开销，我们将帧率从 60fps 降低到 30fps，然后再次提高到 60fps，每 6 帧切换一次，这比通常的调整间隔要短得多。我们通过每秒采样一次 CPU 处理时间和应用程序私有字节来测量云游戏应用程序和编码器的 CPU 和内存利用率，使用 `typeperf`[228]。我们测量 30 分钟以消除随机性。我们将稳定的 60fps 帧率（`stable`）和频繁切换帧率（`switch`）进行比较，结果见图 6.18。对于 CPU 利用率，两种情况的分布都在 0% 到 20% 之间。`switch` 比 `stable` 稍好一些，因为生产较低的帧率对游戏应用程序的 CPU 资源消耗较少。对于内存利用率，主要的内存消耗来自游戏应用程序。帧率切换会略微增加云游戏应用程序的内存利用率。尽管如此，即使在最差的情况下，在 99 百分位上，帧率切换也只会增加内存利用率不超过 1.8%。这是可忽略的，况且在正常帧率调整下可能更低。

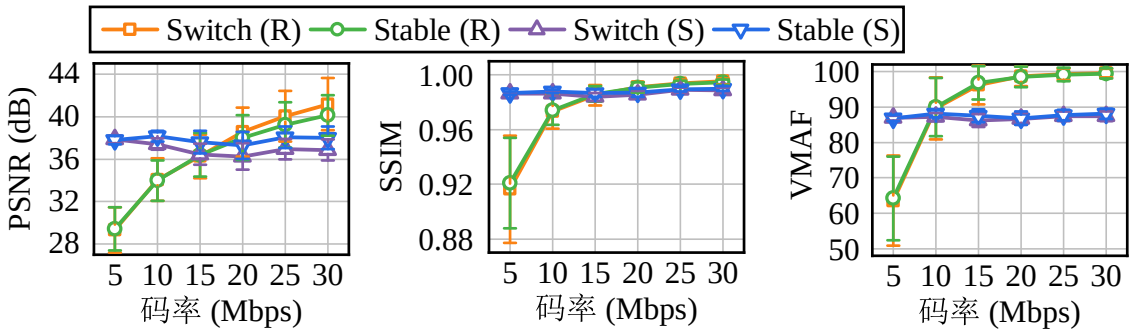


图 6.19　AFR 和原始视频在不同场景下的画质区别

6.6.4.3　潜在的画质下降

我们还研究了 AFR 可能导致的画质下降。我们从游戏中录制两个原始视频，一个是跑动场景（R），另一个是静止场景（S）。对于每个视频，我们每 100 帧切换一次帧率，共切换 15 次，然后测量接下来的 400 帧的视频质量。我们研究了三种视频质量指标，峰值信噪比（PSNR）[31]，结构相似性指数（SSIM）[30]和视频多方法评估融合（VMAF）[115]，并在图 6.19中呈现结果。误差线表示标准差。`stable` 和 `switch` 分别表示帧率保持不变或频繁切换的情况。结果表明，频繁切换帧率不会影响视频质量：三种指标下两个视频的视频质量都是相似的。

6.6.5　现网部署

最终，我们在腾讯云游戏服务的 Windows 客户端上部署 AFR，以评估其性能。在部署 AFR 之前，该云游戏服务遵循 WebRTC 中的帧控策略（即 `DropTail`）。为了进行干净、受控的比较，我们只在生产集群中呈现了在线 A/B 测试的结果，并保持所有其他实现和设置不变。A/B 测试从 2021 年 1 月 8 日至 2021 年 1 月 14 日进行，共有 5369 个以太网会话和 1467 个 WiFi 会话。AFR 的参数设置与模拟（§6.5.2）中保持一致。我们随机为每个会话启用（或禁用）AFR，概率为 50%。我们在表 6.2中呈现结果。对比的指标是第 99 百分位的排队延迟（Q99），延迟大于 50ms 的帧比例（Q>50ms），第 99 百分位的总延迟（T99）和延迟大于 100ms 的帧比例（T>100ms）。卡顿比是卡顿比例大于 5% 的会话比例。类别 (1) 和 (2) 分别是 Windows 客户端的有线网和 WiFi。与仿真结果相似，以总延迟（P(T>100ms)）为指标的卡顿帧比例（§6.6.1）在两个类别中都提高了 34% 和 30%，这显着改善了交互式流媒体的用户体验。卡顿会话（与图 6.14(a)中的相同指标）平均减少了 17%，表明这些用户可以从卡顿流媒体体验中获得缓解。因此，现网部署也证明了 AFR 对高质量实时多媒体用户的显着好处。AFR 已经部署到腾讯云游戏服务的所有生产集群上超过一年，每天服务数千个用户。

表 6.2　现网部署的性能

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

6.7　讨论

本节讨论 AFR 潜在的缺陷。

应用场景。本章主要评估 AFR 在一个云游戏服务的数据集或生产集群上的性能。但是，正如我们在 §6.1和 §6.2中介绍的，解码器队列的过载通常存在于许多高质量的实时音视频场景中，例如 VR 流媒体或 4K 直播，只要它们将高帧率和高比特率的视频流式传输到商业客户端。我们使用云游戏评估 AFR，因为可以访问真实世界的数据集和腾讯 START 生产服务。我们将 AFR 部署到其他场景作为我们未来工作的一部分。

多个控制回路的共存。在实时音视频系统中，还有其他控制回路同时工作。例如，底层拥塞控制器还将根据网络状况控制视频的比特率[60]。视频编解码器还将根据场景调整量化参数[216]。正如我们在 §6.3.2中讨论的，这些参数受不同原因的影响（网络拥塞，解码器退化，场景变化），这些原因是彼此正交的。因此，帧速率的调整与实时音视频系统中的其他控制器正交。在 §6.6.5中，我们评估了 AFR 在我们的真实生产中的性能。我们将不同控制器之间的协调作为未来工作的一部分。

6.8　本章小结

在本文中，我们提出了一种新的基于端到端的控制机制，即自适应帧率控制（AFR），用于减少解码队列的排队延迟，从而提高实时多媒体传输的用户体验。AFR 通过动态调整帧率来减少解码队列的排队延迟。AFR 引入了一个静态控制器和一个瞬态控制器，分别用于减少静态的高负载和突发的到达和服务。我们进一步通过真实数据集驱动的仿真和在生产集群中的部署来评估 AFR 的性能。实验表明，AFR 可以显着减少卡顿比和尾部总延迟。

第 7 章　数据通路传输层：冗余与重传协同的丢包恢复机制

7.1　本章引言

传输层的一个主要挑战是在互联网瞬时高丢包率的环境下，控制视频帧的截止时间的错过比例。由于视频帧之间的空间依赖性和视频帧之间的时间依赖性，实时多媒体期望数据包能够可靠地传输[227]。然而，我们在生产环境中部署的边缘云游戏服务中的测量结果表明，会话可以在瞬时丢包率很高的情况下运行。尽管平均丢包率很低，例如通过适当的码率控制机制，我们的测量结果表明，超过 2% 的视频帧的瞬时丢包率高达 20% 或更高。这表明这些丢失的数据包集中在几帧上。因此，尽管边缘部署可以使网络往返时延非常低，但丢失的数据包的重传需要额外的时间，从而会违反截止时间。因此，优化丢包恢复机制以控制视频帧的截止时间错失率（Deadline Miss Rate，DMR）是非常重要的。

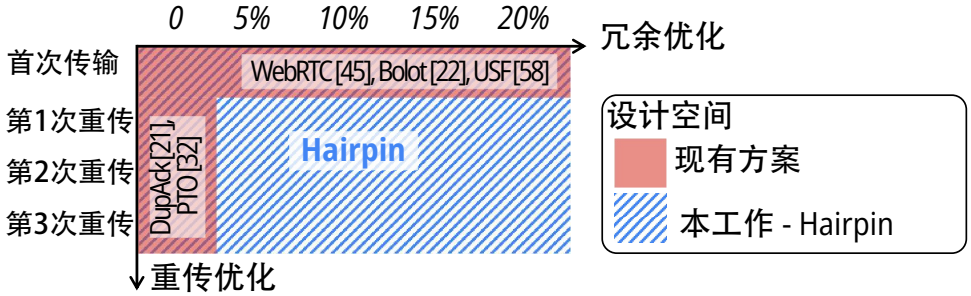


图 7.1　现有解决方案和 Hairpin 的设计空间

不幸的是，现有的解决方案无法在合理的带宽成本下满足 DMR 的严格要求。现有的解决方案分别针对重传机制和数据包的冗余进行优化，或者在应用层添加数据包的冗余。如图7.1所示，一条研究线（垂直方向）致力于从传输层快速重传丢失的数据包，例如 PTO[54]。然而，仅重传丢失的数据包无法满足互动流媒体的要求，因为 DMR 远高于 0.1%（§7.4.3）。另一条研究线（水平方向）致力于自适应的前向纠错（Forward Error Correction，FEC），以便客户端可以基于冗余数据包而不是重传来恢复丢失的数据包[64]。然而，基于冗余的解决方案会带来显著的带宽成本，因为瞬时丢包率很高。对于内容提供商来说，这样高的带宽成本会大大增加运营成本，并降低用户的视频质量。我们所知道的是，现有的解决方案**没有协同优化**重传和冗余。即使采用了现有的解决方案，这种冗余和重传的正交设计仍然无法满足互动流媒体的带宽成本和 DMR 的要求。

本章的主要观察是，我们可以有效地通过利用边缘部署所提供的新的设计空

间来打破这种权衡。边缘部署的互动流媒体服务可以在应用服务器和用户之间实现 10-20ms 的平均往返时延[90,196,225]。在这种情况下，有限次但不太多的重传对于截止时间为 50-200ms 的应用来说是可以容忍的。我们不需要在初始传输中消耗太多带宽来保护数据包免受丢失：我们可以利用潜在的重传，并为需要重传的丢失的数据包添加冗余，以便在截止时间内传输数据包，如图7.1的右下角的区域所示。这带来了新的变化，可以同时减少带宽成本和 DMR（§7.2.3）。由于初始传输期间重传数据包的数量远远少于数据包，因此我们可以显著减少带宽成本。此外，我们还可以利用重传数据包的冗余来减少 DMR。因此，通过在传输层协同设计冗余和重传，Hairpin 能够打破现有的带宽成本和截止时间错过比例之间的权衡，从而实现低带宽成本和低 DMR 的实时多媒体服务。

一个直接的解决方案是设计一个 RTT 感知的自适应启发式 FEC：当 RTT 很高时，增加冗余率；当 RTT 很低时，减少冗余率。然而，这样一个启发式的方案并不是最优的，原因如下（§7.3.2）。首先，优化具有复杂的时间依赖性：一个数据包集的冗余决策可能会**级联地**影响后续数据包的决策，因为需要重传的数据包数量取决于丢失的数据包数量。其次，决策变量（动作空间）和网络条件（状态空间）在许多维度上都很高：它有更多的网络条件（例如 RTT 和瓶颈带宽）和应用程序要求（截止时间），以及每轮传输的更多冗余决策。其次，优化目标是复杂的。即使对于单个视频帧，由于一帧内多个数据包的存在，DMR 和丢包率之间的关系也是超几何分布。此外，不同的应用程序和地区可能对带宽成本和 DMR 的偏好不同。一个特定的手动解决方案可能无法适用于所有场景。

作为回应，本章将重传和冗余的优化问题合并为一个多步规划优化问题，使用马尔可夫决策过程（MDP）。然后，本章提出 Hairpin[①]，一种新的边缘互动流媒体的数据包丢失恢复机制，用于协同优化数据包重传和冗余 (§7.3.3)。我们显式地将多轮传输后的 DMR 和带宽成本期望值表示为马尔可夫决策过程。我们将具有时间依赖性的多步规划问题建模为马尔可夫链中的边，其中马尔可夫决策过程可以有效地优化问题[230]。我们将网络条件编码为马尔可夫链中的节点，并将状态空间减少到两个维度。我们进一步将 DMR 和带宽成本的组合作为我们的优化目标，以满足各种应用程序的需求，并显式地将目标导出为马尔可夫决策过程以实现最优结果。

本章还在基于边缘的云游戏服务中进行了一周的逐包测量活动，以驱动 Hairpin 的设计 (§7.2.2和 §7.2.3)。然后，我们实现了 Hairpin 并使用真实世界的部署进行了评估 (§7.4.1)。实验表明，Hairpin 可以显著推进现有算法的帕累托边界[231]，能够

① 在羽毛球中，Hairpin 指的是停顿放网–选手故意等到羽毛球快落地时（击球的截止时间）再回球[229]。

减少 67%-80% 的截止时间错失率，同时有与最先进的基线相当的带宽成本 (§7.4.3)。在一个在线的云游戏服务中部署 Hairpin 也在不同类型的网络中显示出显著且一致的性能改进 (§7.4.6)。我们将发布代码和测量活动的跟踪数据。

本章的主要贡献如下：

- 我们驱动了边缘互动流媒体服务的设计，通过结合重传和冗余的优化，以减少 DMR 和带宽成本这一需求 (§7.2)。
- 我们展示了在联合优化重传和冗余的边缘互动流媒体服务中的挑战，并提出了 Hairpin，使用马尔可夫决策过程进行建模 (§7.3)。
- 我们在一个云游戏应用程序中实现并集成 Hairpin，并使用跟踪驱动的仿真和真实世界的部署进行了广泛的评估 (§7.4)。

7.2　观察与动机

我们首先介绍我们在丢包上的测量结果（§7.2.1），分析现有解决方案的不足（§7.2.2），并驱动 Hairpin 的设计（§7.2.3）。

7.2.1　边缘实时流媒体的丢包问题

我们从网络中的观察是，虽然平均丢包率只有 10^{-3}，但**瞬时**丢包率可能非常高。我们在每天有数万用户的生产环境中测量了一个星期的云游戏服务，并计算不同时间尺度的丢包率。具体来说，我们首先计算**会话级丢包率**，即一个用户会话中（可能几分钟到几小时，包含至少数万帧）丢失的总数据包数的比率，以反映长时间尺度上的平均丢包率。然后，我们计算**帧级丢包率**，即一个帧（几十毫秒）内丢失的数据包数的比率，以显示短时间尺度上的瞬时丢包率。例如，如果一个会话有 100 万个数据包，其中有 10 个丢失，那么会话级平均丢包率为 0.01%。同时，如果这 10 个数据包属于同一个视频帧，该帧有 50 个数据包，那么对于这一帧帧级丢包率为 20%，对于其他帧帧级丢包率为 0%。

如图 1.3所示，会话级丢包率中位数为 0.05%，与类似的测量[232]相当。然而，瞬时帧级丢包率可能非常高：2% 的帧在一个帧内丢失了 20% 以上的数据包。这样高的瞬时数据包丢失对于将 DMR 控制在 10^{-3} 或更低水平来说是一个巨大的挑战 — 我们不能再忽略这些瞬态行为，而是必须即使在瞬时丢包率很高的情况下也要及时传输视频帧。

同时，这些丢包不能轻易通过降低发送速率来缓解。为了实现低延迟，大多数拥塞控制算法在交互式流媒体中使用**延迟**作为减少发送速率的信号（例如，BBR[57]，Copa[49]，GCC[60]）。在这种情况下，由于发送速率已经提前根据增加的延迟而减

少，因此不会发生拥塞丢包，这也已经在相关工作中得到了证实[60]。我们的在线测量也揭示了类似的观察结果：我们的云游戏服务已经采用了类似于 GCC[60] 的基于延迟的拥塞控制算法，该拥塞控制算法被广泛部署在交互式流媒体应用程序中，例如 Chrome 和 Stadia。我们进一步证明了 RTT①和带宽之间的弱相关性（§7.4）。如图 1.3所示，我们的测量显示，随着 RTT 的增加，丢包率并不会显著增加。这表明，仅控制比特率或帧率仍然不足以避免瞬时丢包。因此，我们需要一种新的方法来控制瞬时丢包率，以便在瞬时丢包率很高的情况下也能及时传输视频帧。

7.2.2　现有机制不足之处

如我们在 §7.1中讨论的，丢包对于边缘实时多媒体的 DMR 贡献很大。因此，我们研究了现有的丢包恢复机制为何无法满足边缘实时多媒体的要求。现有的解决方案主要分为两类，如下所述。

基于重传的机制. 现有的传输协议（例如 TCP）依赖重传来应对丢包。仅依靠重传是无法实现 DMR 为交互式流媒体帧的 0.1% 或更低的极低水平的。例如，当数据包丢包率瞬时为 20% 时，即使进行 3 次重传，仍然会有 0.16% 的数据包丢失。请注意，由于每个帧可能有几十到几百个数据包，因此即使丢失一个数据包也会违反该帧的截止时间要求，因为实时多媒体需要可靠地传输所有数据包（§7.2）。因此，即使依靠重传和速率控制，帧的 DMR 仍然非常高。第7.4节的评估也证明了性能下降。

基于冗余的机制. 还有一些解决方案使用冗余机制，例如 FEC。然而，现有的自适应 FEC 解决方案（来自工业界[51,64]和学术界[62-63,66]）仅针对初始传输优化 FEC 参数。它们根据丢包率调整 FEC 包的数量，并在发生数据包丢失时重新传输数据包。请注意，数据包丢失并不是确定的：当瞬时丢包率增加到 20% 时，它并不意味着每五个数据包就会正好丢失一个数据包。在这种情况下，为了实现 DMR 为 10^{-3} 或更低的极低水平，FEC 率需要远远高于丢包率，从而导致很高的带宽成本（§7.4）。例如，WebRTC 是一种最先进的交互式流媒体框架，它将 100% 的 FEC 包与数据包一起发送，以便在丢包时能够立即恢复。因此，即使依靠重传和速率控制，帧的 DMR 仍然非常高，带宽成本也很高。我们在 §7.4中的评估也证明了包含 WebRTC 在内的其他基线方法的性能下降。

① 在本文中，我们使用 RTT 来表示不包含重传时间的网络层延迟。我们使用应用程序延迟来表示包含重传时间的应用程序层延迟。

7.2.3　动机

因此，在 RTT 降低的情况下，尤其是对于边缘交互式流媒体，重传在一定程度上是可接受的。此时，在如何重传以及重传什么这两个问题上，本文有如下观察：

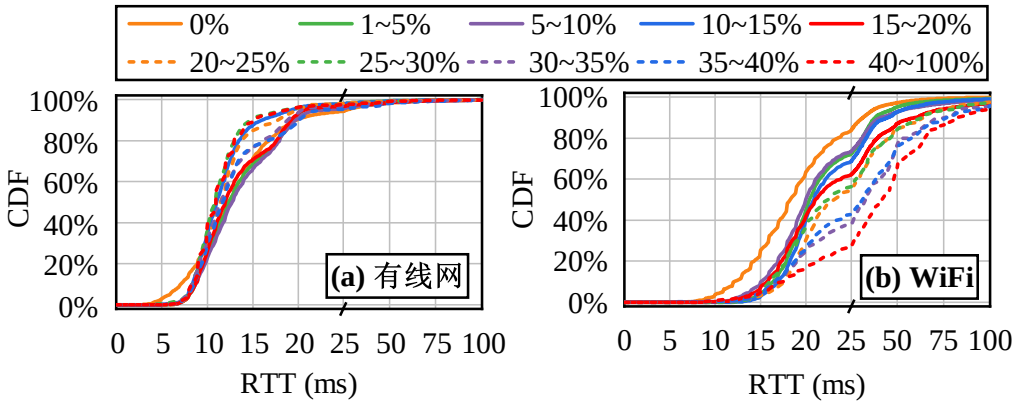


图 7.2　生产环境中按照帧级丢包率分类的 RTT 分布

（1）RTT 远低于截止时间，使得冗余和重传的联合优化成为可能。如我们之前讨论过的，RTT 为 10-20 ms，截止时间为 50-150 ms，多次重传是完全可以接受的。这使得冗余和重传的联合优化成为可能，从而带来两方面的好处：

- 降低截止时间未能满足的比例。在现有的 FEC 机制中，许多截止时间未能满足的情况来自于重传中的数据包丢失。当在重传数据包上添加冗余数据包时，我们可以有效地避免重传数据包的丢失，从而进一步降低截止时间未能满足的比例。
- 减少带宽成本。为了实现相同的 DMR，添加冗余到重传数据包的带宽成本要比仅添加冗余到初始传输的带宽成本要低得多。这是因为重传数据包在带宽消耗中总是少数 – 冗余重传数据包只会带来很小的带宽成本，但却能带来显著的 DMR 改进。

尤其是当多轮重传被容忍（例如，RTT 较小）时，联合优化将具有更大的优势（稍后在 §7.4.4中展示）。因此，我们有动机利用边缘部署所提供的重传机会，联合优化冗余和重传机制。

（2）服务器端的丢包恢复自适应是可能的。动态优化高瞬时丢包率的尾部情况需要快速的自适应。根据我们的测量，服务器和客户端之间的反馈循环小于丢包事件的持续时间，使得冗余和重传的联合优化成为可能。这主要有两个方面：

- 控制回路不会随着丢包率的增加而增加。我们测量了一些云游戏服务的 RTT，并将其分为不同的帧级丢包率区间。如图7.2(a) 所示，帧级丢包率的分布与

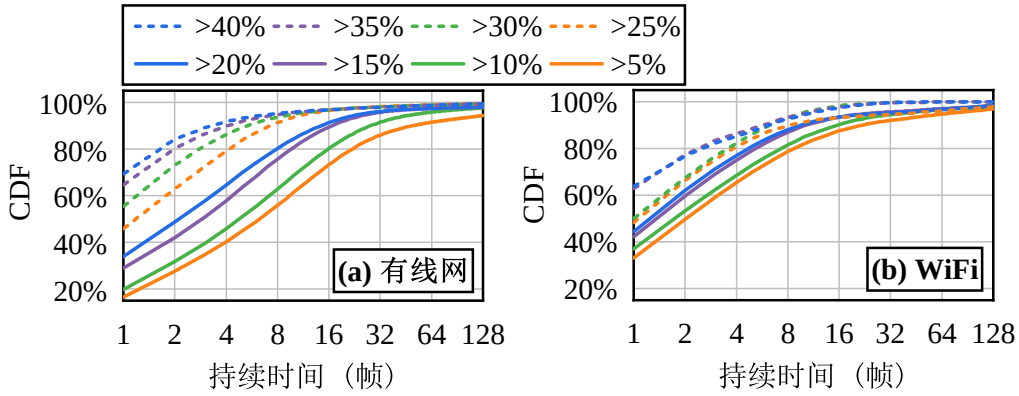


图 7.3　生产环境中的丢包事件持续时间分布

RTT 的分布没有显著的变化。注意，这里的重传包 RTT 不计入丢包率的统计中。WiFi 中的 RTT 随着帧级丢包率的增加而增加（例如，由于链路层的重传[233]）。然而，即使当瞬时丢包率达到 30%（图7.2(b) 中的虚线绿色曲线）时，60% 的已确认数据包的 RTT 仍然小于 25ms。这表明（i）服务器能够快速检测到网络状况的变化，（ii）即使在瞬时丢包率增加时，仍然有多次传输机会。

- 丢包事件的持续时间是短暂的，但仍然比几个控制回路的时间长。我们测量每个丢包率大于不同阈值（5%，...，40%）的时间，如图7.3所示。丢包率是按帧级测量的。网络类型来自腾讯云游戏客户端。根据我们的测量，大多数丢包事件都持续几个 RTT。例如，在帧级别丢包率大于 10% 的情况下，大约有 70% 的丢包事件持续时间大于 2 帧。当帧率是 60fps 时，这已经是 RTT 中位数（12ms）的几倍了。因此，服务器的反应仍然有效，其仍然可以通过调整冗余参数来缓解丢包。

7.3　冗余-重传联合优化器设计

FEC 包含两个参数 (d,k)，其中 d 个数据包和 k 个冗余包作为一个块发送。如果在 FEC 块 (d,k) 中最多有 k 个包丢失，理想的 FEC 解码器可利用剩余的包恢复所有数据包[234-236]。我们用 $\beta=\frac{k}{d}$ 表示为 FEC 冗余率，d 表示 FEC 块大小。Hairpin 中的决策变量是每次（重新）传输时的冗余率 β 和块大小 d。

如我们之前讨论过的，基于边缘的交互式流媒体需要降低 DMR 和带宽成本。为了清晰起见，我们首先给出帧截止时间丢失率（DMR）和带宽成本（Bandwidth

Cost，BWC）的公式：

$$DMR = \frac{\text{截止时间后到达的数据帧}}{\text{总数据帧}}$$
$$BWC = \frac{Redundancy_{byte} + Retransmission_{byte}}{Data_{byte}} \tag{7.1}$$

较高的 DMR 或 BWC 意味着更频繁的卡顿或更高的运营成本，这两者都是实时多媒体服务提供商尽量避免的。

在本节中，我们首先总结一些设计直觉，然后提出一个简单的解决方案（§7.3.1）。然后，我们讨论了重传和冗余的联合优化的设计挑战（§7.3.2）。我们通过提供一个基于马尔可夫链的优化算法来解决这些挑战，以有效地改善 DMR 和 BWC（§7.3.3）。最后，我们在 §7.3.4中讨论了 Hairpin 如何处理测量中的不准确性、在线部署中的开销以及其他实际问题。

7.3.1　基本想法和简单的方案

基本想法：基于多次重传机会规划的自适应冗余比例。边缘实时多媒体的短 RTT 使得数据包有多次重传机会而不违反截止时间。RTT 和剩余时间 t 的比率表示潜在的重传次数。例如，当当前 RTT 为 20ms，数据包仍有 40ms 的截止时间时，比率为 $\frac{t}{RTT} = \frac{40ms}{20ms} = 2$，表示这些数据包在截止时间前大约可以重传两次。具有更多重传机会的数据包可以更好地利用潜在的重传来在截止时间前传输数据包，这已经在 §7.2.3中讨论过。因此，我们的基本想法是在优化冗余率时考虑未来的重传机会。当一批数据包有更多可预见的重传机会（即截止时间仍然很远）时，我们可以减少冗余率以节省带宽成本。当这些数据包的剩余时间接近截止时间时，我们可以增加冗余率以降低 DMR。

简单的方案：基于网络 RTT 的冗余率优化。因此，一个简单的方案是（i）在初始传输和重传时都添加冗余，（ii）在优化冗余率时考虑剩余的传输机会。由于已经有基于网络条件的冗余率优化方案[62-64]，我们可以在这些方案的基础上引入一个因子，即基于剩余传输机会的冗余率优化因子，即：一个简单的方案是当传输机会多时减少冗余率，当传输机会少时增加冗余率。因此，我们可以通过引入一个因子来增强这些算法。

具体地，给定丢包率 α 和比特率 B，假设已经确定了 $\beta_0(\alpha, B)$ 应该是现有算法优化后的冗余率。我们可以根据剩余传输机会 $\frac{t}{RTT}$ 来增加或减少冗余率 $\beta_0(\alpha, B)$，即：

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

其中，k 是一个系数，用于在简单方法基础上调整冗余率的增加或减少的积极程度。

事实上，根据 §7.4.5中的评估，这样一个简单的解决方案就足以推动 DMR-BWC 的帕累托边界向前推进。然而，这样一个简单的解决方案仍然面临一系列缺点，这些缺点阻止运营商进一步提高性能。我们将在下一节详细讨论这些挑战。

7.3.2　设计挑战

尽管我们展示了一个基于 RTT 的自适应冗余率算法，但由于以下原因，仍然很难优化这些参数。

7.3.2.1　时间依赖性：多次重传机会的决策级联

当考虑多次传输机会时，一次传输的 FEC 参数决策将级联地影响下一轮传输的优化。例如，如果我们激进地为一组数据包添加高冗余率，则将减少数据包丢失的数量。相反，对于同一组数据包，低冗余率将会在相同的网络条件下概率性地增加数据包丢失的数量。然而，这些数据包丢失会在下一轮传输中带来更多的数据包重传。如果我们考虑未来 L 轮传输的 F 个数据包的所有动作，动作空间将非常大：由于对于每个冗余决策，都有 F 种可能的下一轮传输的数据包数量（取决于有多少数据包丢失），我们需要优化的变量数量将是 $O(F^L)$[①]。因此，在多次重传的这一过大的动作空间中，使用传统的优化方法（如整数规划）在非常大的动作空间中是不切实际的。我们需要协调不同传输轮次中的选择以实现最佳性能。

7.3.2.2　空间依赖性：冗余率和块大小紧密耦合

即使在单轮传输中，不同的变量（例如冗余率，块大小等）仍然与彼此有复杂的依赖关系。这涉及到以下方面：

(1) 单轮中要传输的数据包数量会影响冗余比例。在不同的传输轮次中，要传输的数据包数量是不同的，这取决于上一轮传输中有多少数据包丢失。因此，根据需要重传的数据包数量，冗余率的在带宽成本上的惩罚也会随之变化。例如，当需要重传的数据包较少时，如 §7.2.3中所讨论的，即使为重传添加 100% 的冗余也不会消耗太多带宽。因此，需要重传的数据包越少，冗余率就越激进。这个简单的解决方案并不知道这种依赖关系，因此导致了它的次优结果。

(2) 当使用较大的块时，块的分散可能会导致错过截止时间。由于瓶颈链路的带宽限制，同时发送的数据包可能会被分散[237]，并逐个到达接收方。在这种情况

① 对于一个帧有 50 个数据包的（F=50），并且有 5 次传输机会（L=5，例如 RTT 为 20ms，截止时间为 100ms）的情况，这将变成 10^8 个变量。

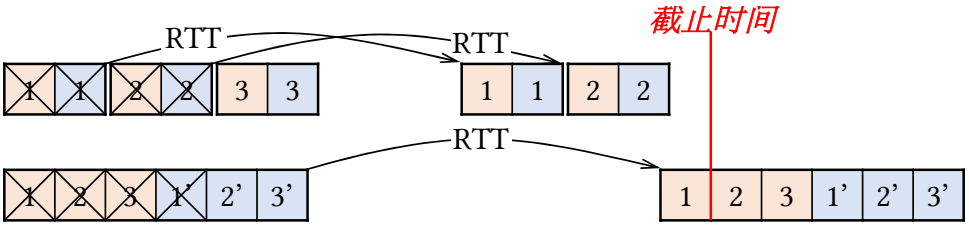
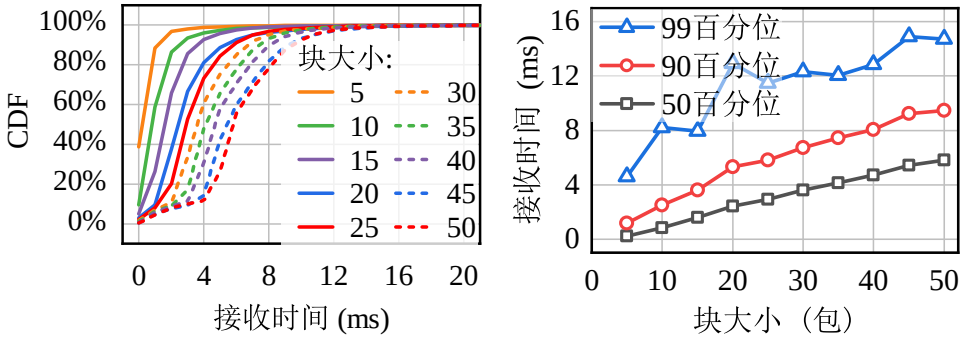


图 7.4　一帧中更小的 FEC 块可能会有更好的性能的示意图



(a) 累积分布　　(b) 不同百分位的趋势

图 7.5　不同块大小的块接收时间

下，构建较大的块将增加等待所有数据包到达接收方的延迟。由于只有在完成一个块后才能确定数据包丢失，因此较小的块可能会更早地知道是否需要重传，并在截止时间之前获得额外的传输机会。例如，在图7.4中，由于早期确定了数据包丢失，因此小块的数据包重传可能会在截止时间之前到达接收方，而大块的数据包则无法在截止时间之前到达。上面和下面的场景分别代表使用小和大块。数据包和 FEC 包用橙色和蓝色表示。我们通过使用不同的块大小，从我们的服务中在线测量 FEC 块的接收时间来量化块大小的影响。如图7.5和图7.5(b)所示，当块大小为 50 个数据包时，超过 10% 的块在接收方处跨越了 10ms，这甚至与 RTT 相当。FEC 块在服务器端以突发的方式发送出去。图7.5(b)是图7.5(a)的处理结果。测量细节见§7.4.2。此外，当丢包率高于冗余率时，较小的块大小可能也会更有利。如图7.4所示，当传输过程中丢失了前四个数据包时，数据包 #3 仍然可以成功传输，这是小块的情况（图7.4中的上面的情况）。对于较大的块，如果丢包率大于冗余率，则无法恢复任何丢失的数据包。

7.3.2.3　复杂的目标：截止时间丢失率和带宽成本

与延迟或吞吐量我们可以直接测量不同，截止时间丢失率的估计值需要考虑多个潜在的传输轮次。因此，简单的解决方案，没有明确估计该帧是否会错过截止时间，将会得到次优的结果。例如，当一个视频帧有几十个数据包时，即使在独立同分布假设下，丢包率和在单个轮次中传输成功的概率之间的关系是**超几何**的。

考虑多个未来的轮次一起，这种关系只会使截止时间丢失率和网络状况之间的关系更加复杂。此外，一些应用程序甚至同一应用程序在不同的操作区域中可能对截止时间丢失率与带宽成本的偏好不同。某些区域的流量成本可能比另一些区域高，而一些应用程序可能会为用户体验付出一切，而另一些应用程序可能不会。因此，我们需要明确地优化目标，以实现最佳结果。

7.3.3　建模与优化

为解决上述挑战，本工作有如下设计。

（1）将多轮规划中的时间依赖性建模到马尔科夫链的边中。马尔科夫链广泛用于顺序决策过程的优化（例如，强化学习[230]）。通过马尔科夫链，我们可以将两轮（重）传输之间的丢包检测建模为两个马尔科夫节点之间的转换。在这种情况下，只需关注当前状态和下一轮中其潜在状态之间的最佳参数，就可以将邻近节点之间的转换的级联效应分离开来，从而大大减少了动作空间。我们进一步证明，在这样的马尔科夫链中，局部关注邻近节点仍然可以获得全局最优结果。

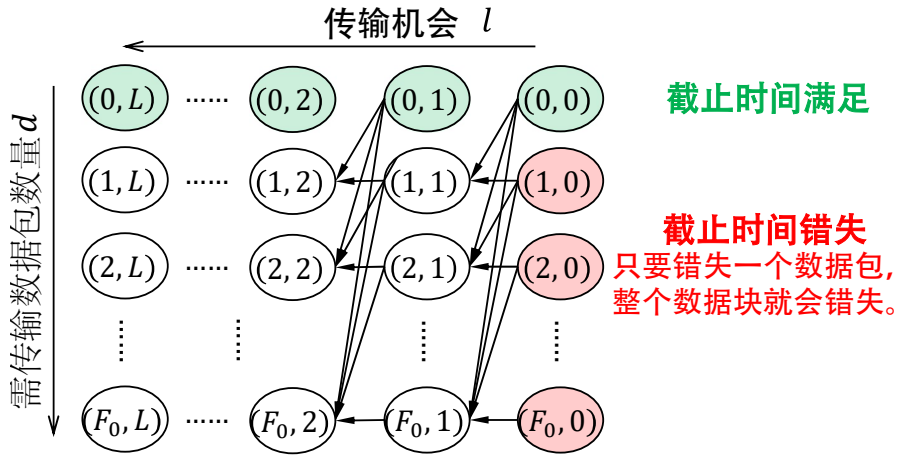


图 7.6　在给定丢包率和帧大小的冗余率优化中的吸收马尔科夫链

（2）将空间依赖性建模到马尔科夫链中的节点中。为了确保考虑了要传输的数据包数量，我们构建了一个二维马尔科夫链，其中两个维度分别表示传输机会和要传输的数据包数量。我们在图7.6中展示了我们的马尔科夫链的状态转移。每个节点由 (d,l) 表示，其中 d 表示剩余要传输的数据包数量，l 表示这些数据包的剩余传输机会。我们的目标是找到节点 (B,L) 的最优冗余率，其中 B 是给定的块大小，L 是来自公式7.3的剩余传输机会。在这种情况下，我们可以将变量之间的时间依赖性和空间相关性都公式化为这个二维马尔科夫链。

（3）使用马尔科夫决策过程显式优化截止时间丢失率和带宽成本。我们最后使用马尔可夫决策过程的形式提供了多轮优化中的截止时间丢失率和带宽成本的

显式表达式。我们从最后一次传输的状态（作为马尔科夫链的最后一层）逆向计算不同状态下的 DMR 和 BWC，直到第一次传输的状态（作为马尔科夫链的第一层）。这样，状态之间的转移概率就可以直接迭代。我们进一步将冗余率和块大小的优化解耦，以提高优化效率。

我们在下面介绍我们的分析模型和算法。在交互式流媒体中，帧是从服务器连续生成并发送出去的。一个流中有数千到数百万个帧，具体取决于特定的应用程序，其中前面帧的重传与后面帧的传输重叠。因此，类似于力学中的有限元分析[238]，我们从流中挑选一个帧，分析该帧的预期 DMR 和 BWC。一个流的预期 DMR 和 BWC 应该与该帧的 DMR 和 BWC 一致。我们在表7.1中列出了将要使用的所有符号。具体来说，Hairpin 通过以下方式优化 FEC 参数：

表 7.1　第7.3节中的符号含义

符号	说明	符号	说明
输入:			
α	网络丢包率	T	截止时间前的剩余时间
RTT	网络往返时延	Θ	网络瓶颈带宽
F	那一帧的帧大小		
中间变量:			
$l(n,r)$	第 r 层在 n 个数据包中的丢包数量	L	剩余传输机会
$k(n,r)$	第 r 层在 n 个数据包中的冗余包数量	DMR	截止时间错过比率
BWC	带宽成本		
输出:			
β_i	第 i 层的冗余率	b_i	第 i 层的 FEC 块大小

步骤 1：计算剩余传输机会。给定当前网络 RTT，截止时间 T，瓶颈带宽 Θ 和某个块大小 d，剩余传输机会 L 可以通过以下公式计算：

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

步骤 2：生成吸收马尔科夫链。然后，我们根据当前的丢包率 α 和帧大小 F 计算给定剩余传输机会 L 的最优冗余率。我们从第 $l-1$ 层迭代地计算吸收马尔科夫链，直到第 l 层。我们将具体的公式推导留在 [239] 中。对于节点 (d,l)，在某个冗余率 β 下，其 DMR 如下：

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

其中，$p((d,l)\to(d',l-1);\beta)$ 是从 (d,l) 转移到 $(d',l-1)$ 的转移概率，可以根据当前的丢包率 α 和冗余率 β 计算。同样，BWC 也可以更新为：

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

其中，后面一项是在第 l 层的额外的 BWC。然后，我们计算 (d,l) 下的最优冗余率 β：

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

并有 $DMR(d,l;\beta_{opt})=DMR(d,l)$ 与 $BWC(d,l;\beta_{opt})=BWC(d,l)$。这里，$utility(DMR,BWC)$ 是一个用于平衡 DMR 和 BWC 的优化目标函数。为了简单起见，我们采用 DMR 和 BWC 的线性组合作为优化目标：

$$utility(DMR,BWC)=DMR+\lambda\cdot BWC \qquad (7.7)$$

注意，Hairpin 并不会落入与基线相同的 DMR 和 BWC 之间的权衡，而是同时提高 DMR 和 BWC，我们将在 §7.4.3中评估。实际上，服务提供商可以根据不同的场景调整系数 λ，以平衡卡顿事件和带宽成本。较低的 λ 表示用户更倾向于 DMR 而不是带宽成本。我们还将在 §7.4.4中评估不同的优化目标函数的性能。

因此，(B,L) 的冗余率可以根据上述公式进行优化。在计算第 l 层的所有节点后，我们可以计算第 $l+1$ 层的 DMR 和 BWC，直到计算出节点 (B,L)。由于节点之间的迭代是线性的，只要 DMR 和 BWC 的效用函数是单调的（例如，线性关系），则优化仍然成立。

我们令 $d>0$ 时 $DMR(d,0)$ 为 1，因为一个丢包就会导致块丢失（用绿色标记）。我们还将所有 $DMR(0,l)$ 设置为 0，因为没有剩余的数据包需要传输。所有这些边界节点的 BWC 均设置为 0。请注意，不同的块大小和剩余传输机会可以复用相同的马尔可夫链来加速优化，因为链只取决于丢包率 α 和帧大小 F。

步骤 3：计算最优块大小。我们从 1 到帧大小枚举可能的块大小，根据步骤 2 中的链计算每个块的 DMR 和 BWC，最后根据给定的效用函数找到最优块大小。我们将数学细节留在 [239] 中。根据我们在 §7.4.5中的评估，不出所料，当瓶颈带宽高（即，数据包不怎么被打散）时，大多数场景的最优块大小是帧大小。然而，当数据包被打散时，构建较小的块可以实现更好的 DMR。运营商可以优化块大小以实现最后一点点的改进。

在块大小的优化过程中，我们还优化了检测到丢包时的权衡，即尽快重传该数据包还是等待其他数据包形成 FEC 块。在恢复能力方面，将多个丢失的数据包

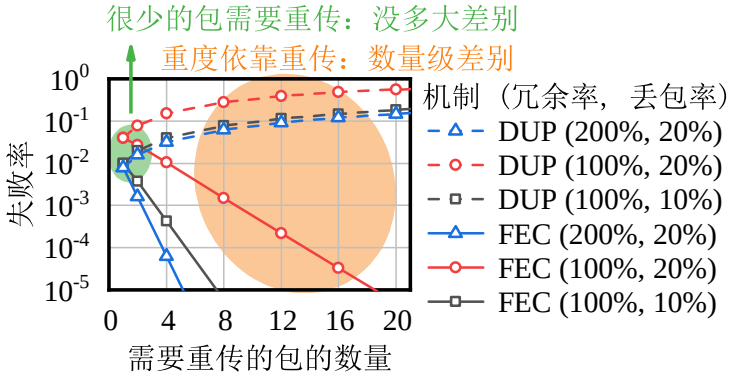


图 7.7　不同的重传策略在不同的丢包率和冗余率下的重传失败率

构成一个 FEC 块可能比单独重传（或重复，如果有冗余的话）每个数据包更有效。我们计算了在不同的冗余率和丢包率下重传这些数据包的失败率，并在图7.7中展示了一个理论上的使用不同的重传策略（逐包重复或构造 FEC 块）在不同的丢包率和冗余率下的重传失败率的例子。随着需要重传的数据包数量的增加，DUP 的重传失败率会增加，因为我们需要确保每个数据包都被传输。当需要重传的数据包较少时，无论是重复还是构造 FEC 块都没有显著差异（虚线和绿色阴影的实线）。然而，当在**尾部**处进行优化以进行交互式流媒体时，一个帧中可能会有多个数据包丢失。因此，考虑到每个帧可能包含数十个数据包，尾部可能会遭受 4 个或更多的数据包丢失。构建这些重传数据包的 FEC 块可以重传失败率降低数个数量级。

步骤 4：获取最优参数。最后，基于网络条件和截止时间，Hairpin 可以根据步骤 3 计算最优块大小，并根据步骤 2 中的块大小计算最优冗余率。

7.3.4　关于部署的讨论

在 §7.3中，我们分析地优化了给定某些网络条件的 FEC 参数。现实可能比理论模型更复杂。在本节中，我们基于我们的运营经验讨论了 Hairpin 的几个实际问题。我们在 §7.4中的基于真实数据的模拟和在生产中的部署也证明了 Hairpin 在线上部署的有效性。

（1）降低线上计算开销。 Hairpin 采用了一种优化算法，这可能无法扩展到生产规模的部署，因为优化需要经常运行（大约每帧一次）并且可以同时扩展到数万个用户。为了解决这个问题，我们希望算法是计算和时间上都高效的。因此我们需要在离线步骤中枚举状态空间并解决每个特定的实例。然后，在在线步骤中，算法将被简化为对预先计算的优化冗余参数的简单表查找。我们将 Hairpin 的状态空间枚举如下。

1. 剩余传输机会：1 到 10。

2. 丢包率：0% 到 50%（以 1% 为单位进行量化）。

3. 帧大小：5 到 60 个数据包（以 5 个数据包为单位进行量化）。

4. 需要重传的数据包数量：5 到 60 个数据包（以 5 个数据包为单位进行量化）。

Hairpin 接下来将每个状态下的最优冗余率和块大小存储在表中。我们发现，更细的量化并不怎么能进一步提升性能。我们在 §7.4.6中的实际评估表明，Hairpin 的内存消耗（2MB）和表查找时间对于在线部署来说是微不足道的。

（2）处理网络波动。我们讨论了 Hairpin 如何处理网络条件的波动。对于 RTT，如图7.2所示，RTT 不会增加太多–无论丢包率如何，中位数 RTT 始终允许 Hairpin 有 3-5 次传输机会。此外，我们还使用短滑动窗口测量 Hairpin 中的网络条件，以确保 Hairpin 具有最新的网络条件。我们将测量窗口设置为 2 帧，并在 §7.4.4中评估此参数的灵敏度。在这种情况下，RTT 的瞬态波动可以立即反映在优化结果中。我们稍后在 §7.4中证明 Hairpin 在真实世界的仿真和生产部署中表现良好。

（3）处理不同的丢包模式。在本文中，当给定某个丢包率时，Hairpin 假设丢包模式是独立同分布的（见公式7.4）。请注意，某个丢包率的持续时间仍然遵循图7.3中的在线测量结果。在实际部署中，使用能够从不同的丢包模式（突发或任意）中恢复的 FEC 编解码器（如[234]），Hairpin 也可以处理不同的丢包模式，因为 Hairpin 只关注一个 FEC 块中丢失了多少个数据包。例如，当一个 FEC 块中有 4 个数据包丢失时，无论这些丢失是连续的还是在块中分开的，只要该 FEC 块中有 4 个额外的 FEC 数据包，客户端就可以恢复这些数据包丢失。因此，Hairpin 不依赖于底层丢包模式的假设，而是依赖于 FEC 块中丢失的数据包数量。数据包丢失可能在几个帧中连续。在这种情况下，由于边缘部署启用的短控制回路，如 §7.2.3中所分析的，Hairpin 应该已经及时地做出反应。

7.4　性能评估

我们在 §7.4.1和 §7.4.2中分别介绍了 Hairpin 的实现和实验设置。我们进一步回答了以下问题：

- Hairpin 如何在基于数据的仿真中表现？我们发现 Hairpin 可以推动基线方法的帕累托边界（见图7.9） (§7.4.3)。
- Hairpin 对于不同的参数设置有多敏感？我们调查了 Hairpin 在不同的参数设置下的性能变化，并证明 Hairpin 在各种参数设置下都有性能提升 (§7.4.4)。
- Hairpin 为什么能够优于其他基线方法？在 §7.4.5中，我们分析了 Hairpin 的性能提升。

- Hairpin 在实际部署中的表现如何？最后，我们在生产服务器中部署 Hairpin，发现 Hairpin 在实际部署中显著改善了 DMR 和 BWC (§7.4.6)。

7.4.1　Hairpin 实现

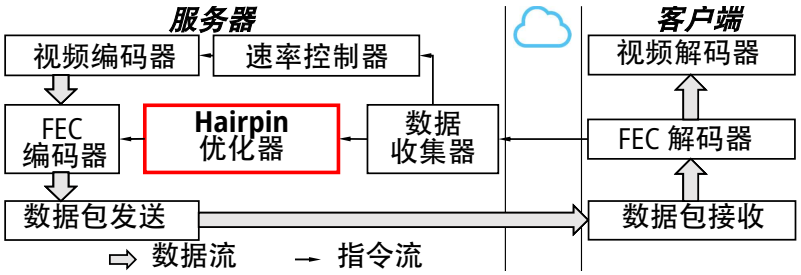


图 7.8　Hairpin 实现总览

我们将 Hairpin 实现为一个基于 `ns3` 的 WebRTC 模拟器[240]的应用层模块，以及在生产环境中的腾讯 START 云游戏应用。Hairpin 在协议栈的工作流程如图7.8所示。没有 Hairpin 时，应用程序首先使用视频编码器对交互式内容进行编码，然后按帧将其发送到传输层。然后，视频帧可以在客户端的协议栈中接收。包发送器和包接收器抽象了传输层的网络栈用于连接管理。然后，视频解码器对流内容进行解码并将其显示给用户。同时，服务器将网络状况（例如，RTT，数据包丢失事件）测量到统计数据收集器中，并将其报告给速率控制器，以适应性地调整流的比特率[60]。Hairpin 插入在现有的应用层和传输层之间，并根据当前的网络状况优化冗余参数，如图7.8所示。我们的云游戏服务中的底层传输协议是基于 UDP 的 RTP 协议的自定义版本[241]，以允许丢失冗余数据包而不修改客户端的内核。我们实现了 Reed-Solomon FEC，因为其在冗余率小于 100% 时的恢复性能[234]，并为当冗余率大于 100% 时实现了自定义的 FEC 编解码器，以便在传输层中使用。注意，Hairpin 也可以和其他编解码器（例如，XORFEC，FlexFEC 等）一起使用，只要它们的参数可以暴露给 Hairpin 即可。

7.4.2　实验设置

数据集. 对仿真数据而言，我们在腾讯 START 云游戏中收集了一个数据集，其中包含了两周的游戏数据，分别是 2021 年 1 月和 8 月，共计超过 1 亿个视频帧和超过 600 小时的游戏时间。这也支持了 §7.2和 §7.3中的测量。用户通过以太网、WiFi 或蜂窝连接访问我们的云游戏服务，我们从云游戏客户端收集这些数据。云游戏服务以 60fps 的帧率进行流媒体，并且比特率在 2Mbps 到 30Mbps 之间。网络状况在我们云游戏服务的服务器上记录，包括平均 RTT、平均比特率和丢包率（大约每 16ms 一帧）。该数据集包含 1995 个以太网游戏会话、741 个 WiFi 会话和

572 个蜂窝会话，每个会话持续几分钟到几小时。据我们所知，我们是第一个在实时多媒体服务中收集在线数据集的人，这些数据集在两周内持续测量，同时在帧级别（大约每 16ms 一帧）记录网络状况。

基线方案. 我们正交地评估了公共的自适应 FEC 机制和重传机制的部署情况。在重传优化的轴上，我们实现了以下基线。

- `Out-of-order`. 传统上，通过检查乱序数据包（例如，TCP 重复 ACK）来检测数据包丢失[242]。我们将其作为我们的默认丢包检测机制。
- `Probe timeout (PTO)`. 此外，为了快速检测尾部数据包的丢失，最近的研究人员还提出了一种基于超时的丢包检测机制[54]。

在冗余参数优化的轴上，我们实现了以下基线。

- $\texttt{WebRTC}_{'14}$. 这是 Google 在 2014 年发布的研究论文[51]中的自适应 FEC 机制。
- $\texttt{WebRTC}_{NOW}$. 这是 WebRTC 现在使用的自适应 FEC 机制（被 Google Stadia[108]、Meet[43]等采用），取代了 $\texttt{WebRTC}_{'14}$。两者的区别是 $\texttt{WebRTC}_{'14}$ 意识到 RTT 并会在 RTT 较低时减少冗余率，而 $\texttt{WebRTC}_{NOW}$ 在添加冗余时更加积极。我们迁移了 Chromium 的 m88 版本的实现，该版本于 2020 年 12 月发布[41]。
- `Bolot`[62]和 `USF`[63]是两个来自研究社区的自适应 FEC 算法。与 Hairpin 不同的是，它们不会为重传添加冗余数据包。
- `RTX` 不添加任何冗余，而完全依赖重传。

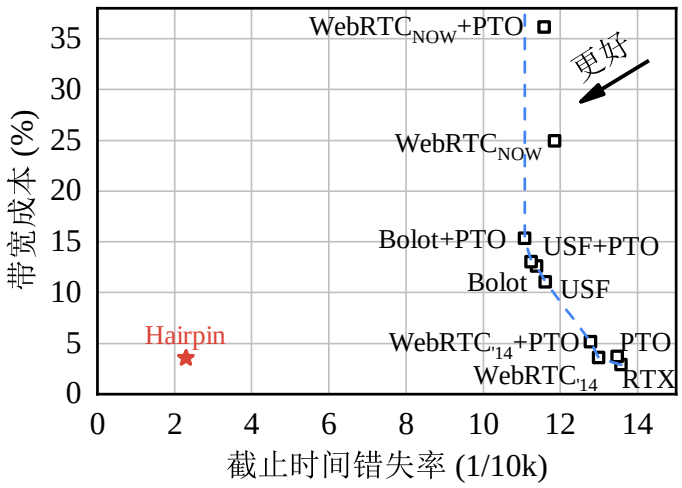
注意到，这些基线都不会优化重传的冗余。由于这两条线路是相互独立的，我们组合实现了 2 (重传) × 5 (冗余) = 10 个基线。

Hairpin 设置. 在我们的仿真中，我们将公式7.7中的效用函数的系数设置为 $\lambda = 10^{-4}$，网络状况的测量窗口为 2 帧，截止时间为 100ms。我们在 §7.4.4中评估了这些参数设置的敏感性。

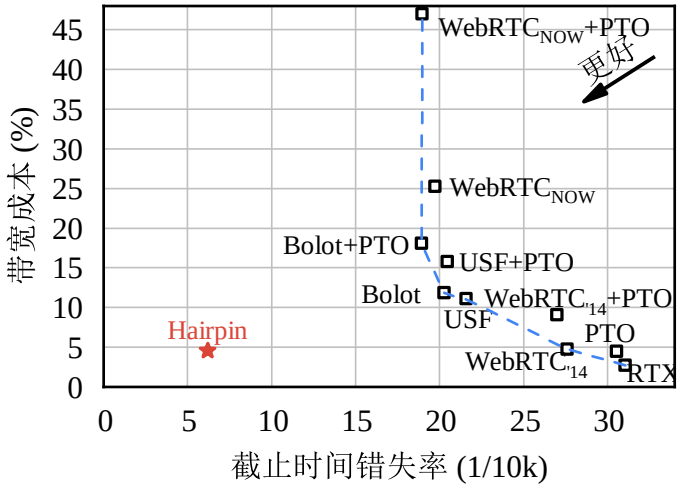
7.4.3　基于数据集的仿真

为了评估 Hairpin 在动态网络状态下的性能，我们在真实世界的跟踪数据上模拟 Hairpin，如 §7.4.2中所述。我们使用 `ns-3` 模拟收集的跟踪数据中的丢包率和 RTT，并评估 Hairpin 是否能够捕获网络动态中的丢包和 RTT 变化，并有效地适应真实跟踪数据。我们首先在图7.9中展示了 DMR 和 BWC 之间的三组跟踪数据的权衡。蓝色虚线是所有基线方法在帕累托边界上的包络线。

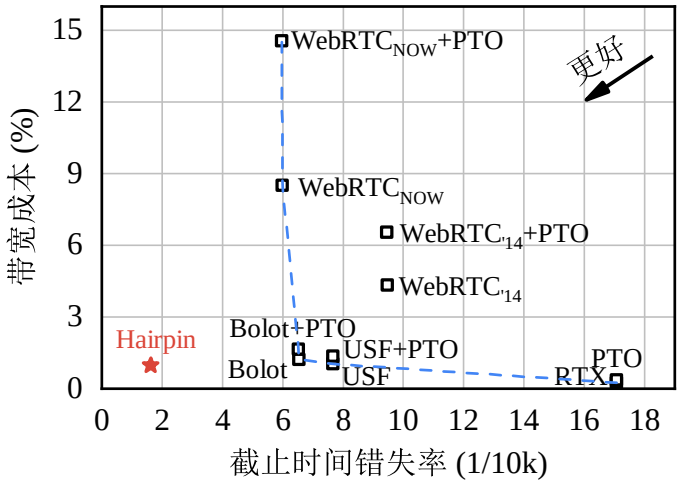
如图7.9所示，`RTX` 的带宽成本最低，因为 `RTX` 只在丢失数据包后才重传数



(a) 有线网数据集



(b) WiFi 数据集



(c) 蜂窝网数据集

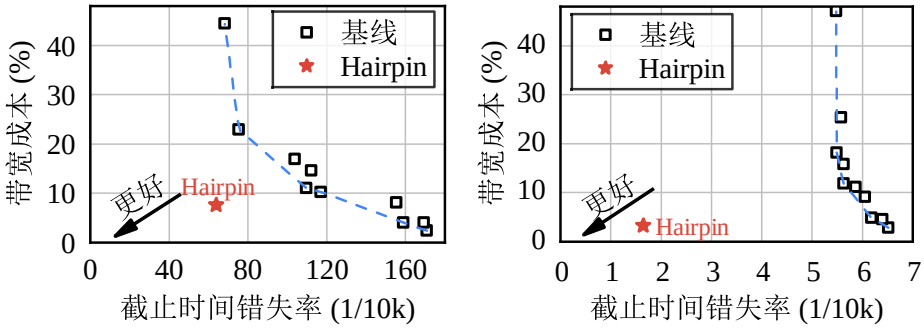
图 7.9　大规模数据集上的仿真

据包。但是，它也是所有基线中 DMR 最高的。同时，$\texttt{WebRTC}_{NOW}$ 与 PTO 一起工作的带宽成本最低，但 DMR 也最低。其他基线都位于 DMR 和 BWC 之间的 Pareto 前沿。相比之下，Hairpin 可以打破这种权衡，并在 DMR 和 BWC 方面实现显著的改进，如图7.9中的红色星号所示：Hairpin 比 DMR 的最低值（$\texttt{WebRTC}_{NOW}$）还要低的 67%-80%，并且与 RTX 的 BWC 相当。因此，如上所述，Hairpin 可以显著提高 DMR 和 BWC，而其他所有基线都无法做到。

注意到，追踪数据是从生产集群中收集的，因此它们包含了网络 RTT 和丢包率的实时变化，具有每 16ms 的细粒度。WiFi 跟踪数据的结果比以太网跟踪数据差，因为 WiFi 跟踪数据的丢包率和 RTT 都更高，如 §7.2.3中所述。在蜂窝网络跟踪数据上的结果令人惊讶地好。这是因为，在我们的在线测量期间，该客户端刚刚开始为蜂窝用户提供云游戏服务，因此在那时对网络状况进行了准入的访问控制。总之，Hairpin 可以显著推进现有基线的帕累托边界，在所有测试数据集中都有很好的表现。

7.4.4　参数敏感性

我们在本节中评估了 Hairpin 的参数对其性能的影响。



(a) 截止时间为 50ms　　(b) 截止时间为 200ms

图 7.10　在不同截止时间要求下 WiFi 数据集上的性能

（1）截止时间的设置。在 §7.4.3中的评估中，截止时间设置为 100ms。我们还研究了 Hairpin 在截止时间较短或较长时的性能。因此，我们在 WiFi 跟踪数据上展示了当截止时间设置为 50ms（图7.10(a)）或 200ms（图7.10(b)）时，Hairpin 和基线的 DMR 和 BWC 的结果。如图7.10所示，给定相同的跟踪数据，当截止时间较短（50ms）时，Hairpin 相对于基线的优势略有减少。这是因为当截止时间较短时，重传机会较少，设计空间也较小。尽管如此，Hairpin 仍然比所有现有的基线都要好得多，尤其是在 DMR 和 BWC 方面。当截止时间较长时，由于重传的设计空间更大，因此优势更大。在其他跟踪数据上的结果也是如此。

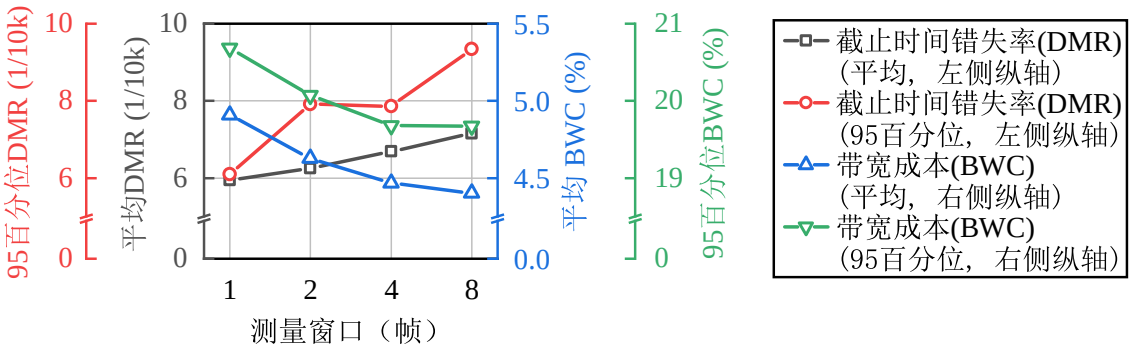


图 7.11　§7.3.4中测量窗口的敏感性分析

（2）测量窗口。我们还通过调整我们在 §7.3.4中讨论的网络条件的测量窗口来评估 Hairpin 的性能。由于 Hairpin 根据实时测量的网络条件来优化冗余参数，因此测量窗口的大小可能会影响 Hairpin 的性能。我们将测量窗口从最近 1 到 8 帧进行变化，并在 WiFi 跟踪数据上重新进行实验。我们测量平均和 95% 分位数的 DMR 和 BWC，并在图7.11中呈现结果。DMR 和 BWC 的结果比较鲁棒：通过将测量窗口从 1 到 8 变化，平均 DMR 和平均 BWC 的变化范围为 0.47%-0.49% 和 6.94%-7.19%，这远低于 §7.4.3中的改进（图7.9(b)）。在实践中，运营商可以根据网络状况的波动情况来决定测量窗口。

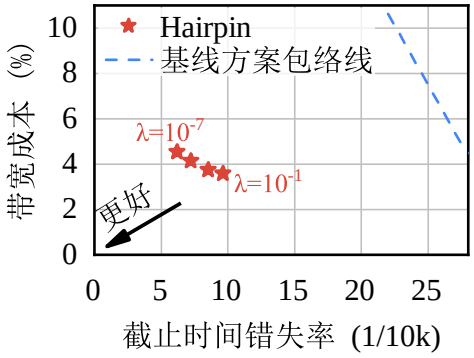


图 7.12　Hairpin 效用函数中 λ 的参数敏感性分析

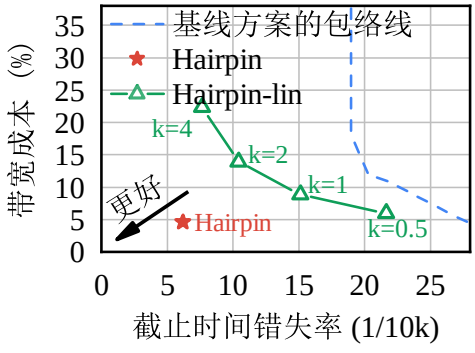


图 7.13　基于启发式算法的 Hairpin (Hairpin-lin)

（3）效用系数 λ。这里还对 §7.4.2中引入的公式7.7中的效用系数 λ 进行评估。它可以调整用户对 DMR 和 BWC 之间的权衡的偏好。较高的 λ 表示用户更喜欢 BWC，而较低的 λ 表示 DMR 的优势更大。因此，我们将 λ 从 10^{-1} 变化到 10^{-7}，并在图7.12中展示了在 WiFi 跟踪数据上，Hairpin 在不同 λ 下的 DMR 和 BWC，注意 Y 轴这里被缩放过了。请注意，图7.12是从图7.9(b)中放大的。如图7.12所示，随着 λ 的减小，Hairpin 的 DMR 和 BWC 都会有所改善。这是因为随着 λ 的减小，Hairpin 会更多地考虑 DMR，而不是 BWC。因此，管理员可以通过根据应用需求调整 λ 来调整 Hairpin 的优化目标。

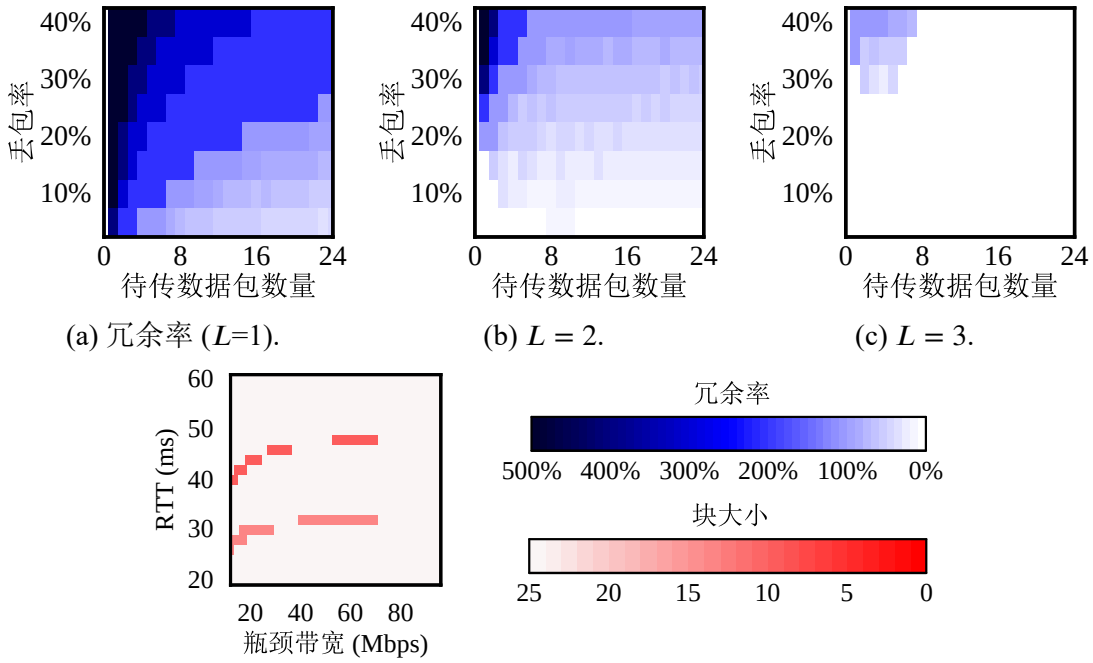
7.4.5　深入理解 Hairpin

我们还在如下方面对 Hairpin 进行了更深入的理解。

7.4.5.1　使用马尔可夫链的改进

我们在 §7.3.2中分析了一个简单的解决方案，但这并不足以充分利用冗余和重传的设计空间。因此，我们还评估了我们在 §7.3.1中提出的启发式基线（Hairpin-lin），用 k 从 0.5 到 4 进行扫描，结果如图7.13所示。基线方案的包络线来自于图 7.9(b)。如图7.13所示，Hairpin-lin（绿线）确实改善了与现有基线（虚线蓝线）的权衡。但是，Hairpin-lin 与基于马尔可夫链的 Hairpin（红星）之间仍然有一半的差距。因此，必须通过使用马尔可夫链来分析问题，进一步打破提升这一权衡。

7.4.5.2　理解 Hairpin 的决策



(a) 冗余率 (L=1).　(b) $L = 2$.　(c) $L = 3$.

(d) 块大小与图例.

图 7.14　Hairpin 的优化结果示意

我们还展示了 Hairpin 的冗余率结果，以进一步了解 Hairpin 在不同情况下如何进行优化。对于冗余率，由于吸收马尔可夫链的优化（§7.3.2）依赖于剩余传输机会 L、丢失率、剩余数据包以及帧大小，我们在图7.14(a)到7.14(c)中展示了不同参数下 Hairpin 的优化冗余率。随着传输机会的增加，Hairpin 会降低冗余率，并依赖重传来恢复丢失的数据包。随着要重传的数据包数量的减少，如 §7.3.2中所讨论的，Hairpin 也倾向于更高的冗余率。此外，当要传输的数据包数量较少时，优

化的冗余率可以高达 500%。这说明了冗余率大于 100% 的有效性。

在对块大小的优化上，如我们在 §7.3.2中讨论的，在许多情况下最佳块大小是帧大小（24 个数据包）。尽管如此，我们也讨论了在剩余传输机会的决策边界上，较小的块大小可以获得更好的性能，因为它们有额外的传输机会。如图7.14(d)所示，尽管最佳块大小在大多数情况下是帧大小，但当 RTT 在 33ms 和 50ms 之间时（1、2 和 3 次传输机会之间的分界点），最佳块大小可能小于帧大小。例如，与将块大小设置为帧大小相比，Hairpin 的优化块大小可以在 RTT 为 50ms 和瓶颈带宽为 60Mbps 的情况下减少 1.78×。我们将对块大小的优化视为如果想要进一步性能提升的手段。

7.4.5.3　Hairpin 的平均延迟

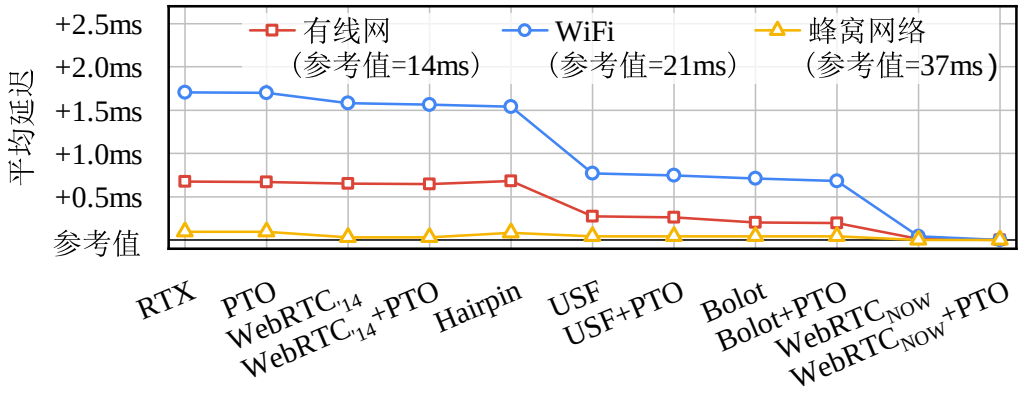
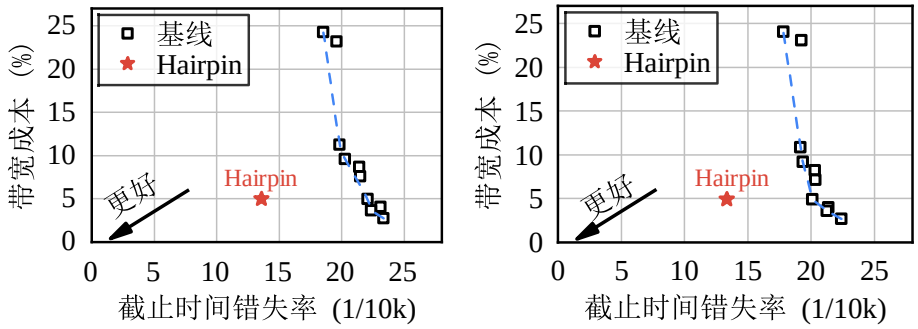


图 7.15　§7.4.3中的平均端到端延迟

除此外，我们还测量了 §7.4.3中 Hairpin 和不同基线的成功传输帧的平均端到端延迟。如图7.15所示，Hairpin 的平均端到端延迟确实增加了，但是与最低平均延迟相比，仅仅增加了 0.1-1.5ms。考虑到我们在 §7.2中讨论的截止时间上的提升优势，这 1ms 对于 RTT（1%-7%）来说是微不足道的。值得指出的是，我们为比较而对齐不同跟踪中的最低平均延迟。此外，运营商也可以采用更保守的映射（例如，增加 λ）来在平均延迟和尾延迟之间进行权衡。

7.4.5.4　和拥塞控制集成

为了进一步研究 Hairpin 与拥塞控制算法的性能，我们在 WebRTC 框架中将 Hairpin 与两个拥塞控制算法，GCC[60] 和 NADA[47]，进行集成。然后，我们通过将它们的带宽，RTT 和丢包率设置为 ns-3 中的链路参数来重放收集的数据集。带宽范围从 2Mbps 到 30Mbps。如图7.16所示，Hairpin 仍然可以在所有现有基线上实现显著的优势。



(a) GCC[240]　　(b) NADA[47]

图 7.16　在不同拥塞控制算法下 WiFi 数据集上的性能

7.4.6　真实世界实验

最终，我们将 Hairpin 部署到我们的云游戏服务中的生产服务器中。我们在生产服务器上与 $\texttt{WebRTC}_{NOW}$ 基线对比进行了一个 A/B 测试。云游戏服务的比特率也符合 §7.4.3中模拟的 2-30Mbps 范围。A/B 测试在 2021 年 9 月进行了一周，共涵盖了 1.7 万个会话，所有会话的持续时间至少为 4 分钟。Hairpin 已经集成到该云游戏服务的 UDP 协议栈中。由于我们在部署 Hairpin 后还部署了其他优化，为了进行公平的比较，我们展示 2021 年 9 月的受控 A/B 测试结果。结果如表7.2所示，其中 P(DMR>1%) 表示平均 DMR 大于 1% 的会话比例。

表 7.2　真实世界实验结果

有线网	DMR	BWC	P(DMR>1%)	会话数量
$\texttt{WebRTC}_{NOW}$	0.34%	30.4%	6.9%	8380
Hairpin	0.23%	3.0%	4.6%	7306
WiFi	DMR	BWC	P(DMR>1%)	会话数量
$\texttt{WebRTC}_{NOW}$	0.72%	31.8%	19.3%	652
Hairpin	0.51%	3.0%	15.3%	613

性能。如表7.2所示，与 $\texttt{WebRTC}_{NOW}$ 相比，Hairpin 能够提高平均 DMR 和平均 BWC。具体来说，对于以太网会话，与 $\texttt{WebRTC}_{NOW}$ 相比，Hairpin 可以提高 DMR 32%，同时将 BWC 减少 40%。对于 WiFi 会话，DMR 和 BWC 的提高分别为 30% 和 43%。我们还测量了平均 DMR 大于 1% 的会话比例，即尾部会话。Hairpin 还可以将尾部会话减少 34% 和 21%，对于以太网和 WiFi 会话，与 $\texttt{WebRTC}_{NOW}$ 相比。请注意，实际实验中的 DMR 略高于模拟中的 DMR（§7.4.3）。这可能是由于其他外部因素（例如用户设备）可能影响 DMR。但是，Hairpin 仍然可以显著提高用户在 DMR 和 BWC 方面的体验，与 $\texttt{WebRTC}_{NOW}$ 相比。

开销。我们还测量了 Hairpin 的开销。如我们在 §7.3.4中所述，为了在线加速优化，我们预先计算了优化的 FEC 参数，并将结果表存储在在线查找中。在我们的表的量化粒度上，它需要 1.98MB 来存储表，这在服务器上是微不足道的。因为表是静态的，可以由所有连接共享。此外，根据我们的测量，查找表的时间总是小于 1ms，这也是微不足道的，因为表是以帧为粒度查找的。因此，Hairpin 的开销是微不足道的。

7.5　工作局限性

交互式流媒体的延迟组件。Hairpin 可以在边缘服务器和客户端之间的流中获得最大的收益，尤其是当端到端网络延迟占据从视频编码器到解码器的总延迟的时候，如图7.8所示。这在交互式流媒体服务中通常是真实的。相关的测量研究也证明了边缘交互式流媒体的网络延迟仍然是瓶颈[17,227]。因此，我们专注于边缘服务器和客户端之间的流的优化。我们在实际网络的部署证明了优化网络延迟可以显著改善用户的体验（请注意，DMR 是端到端测量的）。Hairpin 也可以与其他延迟组件（例如编码，解码等）的优化一起工作，以进一步提高性能。

应用程序的部署。部署 Hairpin 的另一个问题是服务器和客户端都需要修改以支持冗余和重传。之前已经有人实现了 TCP 上的 FEC 机制[243-244]，它需要修改 TCP 协议栈，这对于大规模产品来说不合适。对于必须使用 TCP 进行传输的场景，Hairpin 的部署可能取决于能否修改客户端的 TCP 数据包接收机制。但是，大多数交互式流媒体应用程序采用 UDP 来减少网络延迟[42-43,108-109]。在这种情况下，Hairpin 可以在服务器和客户端的应用程序中实现，这对于大多数应用程序来说是可行的。

7.6　本章小结

本章提出了 Hairpin，一种用于边缘交互式流媒体的数据包丢失恢复机制，用于联合优化冗余和重传。Hairpin 通过真实世界的测量来驱动联合优化，并使用马尔可夫决策过程来优化冗余和重传。基于跟踪的模拟和真实世界的部署表明，与最先进的解决方案相比，这一联合优化可以显著降低 DMR 和 BWC。

第 8 章　数据通路网络层：公平且平滑的队列管理

8.1　本章引言

网内包调度和队列管理是确保相互竞争的网络应用能公平分享网络资源，并尽可能达到其性能目标（如，高吞吐、低延迟）的有效技术。然而，新兴的实时流媒体应用，如视频会议、在线游戏和虚拟现实，遇到了**性能波动**问题。性能波动表现为突然、剧烈的吞吐量下降或延迟峰值，通常是竞争流量的突发到达模式造成的。性能波动会导致具有**高吞吐量和低延迟（Heavy Real-Time，HRT**①）需求的应用程序（如视频会议）出现故障和停顿。事实上，以前的工作表明，仅仅是 200ms 的延迟峰值就会导致应用层需要几秒钟的恢复时间[5]。

麻烦的是，我们发现，许多当前的队列管理方案不仅不能防止性能波动，而且实际上**加剧**了波动。问题源于两个理想属性之间的基本冲突：最大化吞吐量公平性和最小化性能波动。**我们观察到，严格的公平性在存在突发工作负载的情况下会导致高波动，而简单地缓解波动会导致公平性的削弱。**

为了理解公平性和波动之间的冲突，我们考虑了一个例子，如图8.2所示。一个 HRT 视频连接单独运行在住宅网络链路上，此时另一个用户加载网页（例如，`amazon.com`，§8.7.1中对实验设置进行了详细的介绍）。在使用一系列排队规则的实验中，我们看到视频连接在持续 1 秒左右的时间内遭遇了不可接受的（>190ms[245]）帧延迟。一方面，实时多媒体流的性能最差的排队规则是**公平排队**（FQ），而它是有利于公平性的。实际上，FQ 迅速将带宽资源转移到新的 Web 流。实时多媒体流会因此被阻塞住，它需要几个往返时间才能收到足够的信号来调整视频比特率和拥塞窗口。另一方面，实时多媒体流的最佳设置是最不公平的设置，因为它只是为实时多媒体流提供了优势，而不是 Web 流量。

一个直观的解决公平性与波动的冲突的方法是使用一种优先级方案，其中使用精心计算的“权重”来优先处理敏感类别的流量，以避免极端的不公平性。不幸的是，这是不切实际的。首先，通过标记流量（例如，使用 DSCP 位[87]）的方式来实现这种方案是不切实际的，因为它不是激励兼容的②，因为互联网发送者总是能够从将他们的流量标记为高优先级类别中受益。更糟糕的是，如果仅盲目地遵循各种应用程序的标签，其中可能存在的错误标签会使得任何保证性能的设计

① HRT 表示流在同一时间需要高吞吐量和低延迟。例如，视频会议应用程序不仅需要低延迟，还需要提高比特率以获得更好的质量[177]。

② 最近的工作（例如，L4S[246]）使用激励兼容的标签仍然存在实用性和性能问题，本章的后面部分进一步展示了。

失效。其次，管理员不能简单地事先依据类别分配权重，因为流量分布是动态的，很难预测。

上述讨论使得我们寻找一种队列管理方案，该方案平衡了三个存在紧张关系的属性。首先，该方案在长期运行中，遵守传统的流速公平性。其次，该方案可以控制波动，并使实时多媒体流与突发流模式（即 Web 流量）并存。最后，该方案应该是实用的，即它像 CoDel[68]一样是无参数的，而且不需要任何流标记，也不需要特定于应用程序的配置，例如截止时间[247]。

为此，我们设计了 Confucius①，一种无参数的队列管理方案去平衡公平性与波动。在长期运行中，Confucius 保证了竞争流量之间的公平流调度。但是，在短期内，Confucius 则不会突然调整服务率。相反，当新流到达并且必须调整服务率以确保公平性时，Confucius 将权重**平滑地**调整，从而为实时多媒体流提供几个往返时间来检测网络条件的变化，并相应地调整其比特率和拥塞窗口。更具体地说，Confucius 根据简单的指数加权移动平均（Exponentially Weighted Moving Average, EWMA[248]）分配流速，其中**平滑地**调整权重以确保公平性，同时保持波动在可接受的范围内。我们发现，这种方法在公平性和波动之间提供了良好的权衡；在实验中，我们测量了 Web 流量（受益于严格的公平性）的流完成时间（Flow Completion Time，FCT）与实时多媒体流量（受益于平滑）的帧延迟，以了解这种权衡的影响。在基于真实数据的实验测试中，我们发现 Confucius 通常可以将实时多媒体流的帧延迟降低 90%，同时保持与 Web 流量相当的 FCT。

在设计 Confucius 时，我们面临了几个挑战：

（1）实用性。Confucius 是一种基于类的排队方案，它（像许多其他基于类的方案[83,249]）将低延迟流分组到同一个队列中，以避免与填充缓冲区的流量共享队列带来的延迟影响。这就引出了一个问题，即 Confucius 如何可以无参数地正确地对流进行分类，而不使用标签。在 §8.5中，我们说明了 Confucius 如何根据队列占用情况动态地将流迁移至不同的类别：自然占用缓冲区较小部分的流被聚集在一起，而被观察到是填充队列的流与其他填充队列的流共享一个缓冲区竞争。

（2）性能保证。仅仅说 Confucius 平衡了公平性和波动是模糊的，但是如何将这种模糊的描述转化为严格的服务模型是很困难的。通过对 Confucius 使用的 EWMA 函数进行数学分析，我们计算了几个应用程序类别的性能边界。我们表明，短的以 FCT 为指标的流（如 Web 流量）在我们的设置中相对于公平排队最多会有 360 ms 的最大延迟；实时多媒体流（如实时视频）相对于公平排队会减少 90% 以

① 在本文中，我们将 Confucius（孔子，哲学家）的教育哲学用于流量管理：孔子对学生因材施教，我们对流量因需调度。

上的停顿，而长期的批量传输相对于公平排队不会有任何降低（在稳态情况下）。

（3）避免振荡。在动态环境中，多个控制系统（例如拥塞控制，比特率自适应）同时运行，强制公平性和一致性是危险的。队列管理中的微小变化可能会对应用程序造成巨大的副作用。通过共同谨慎地分配每个队列的服务速率和每个队列的流，Confucius 避免了冲突的决策，这些决策对稳定性是有害的。更重要的是，Confucius 的控制是战略性地缓慢移动的，从而有效地为其他控制系统（尤其是拥塞控制）留下足够的时间来最优地反应。

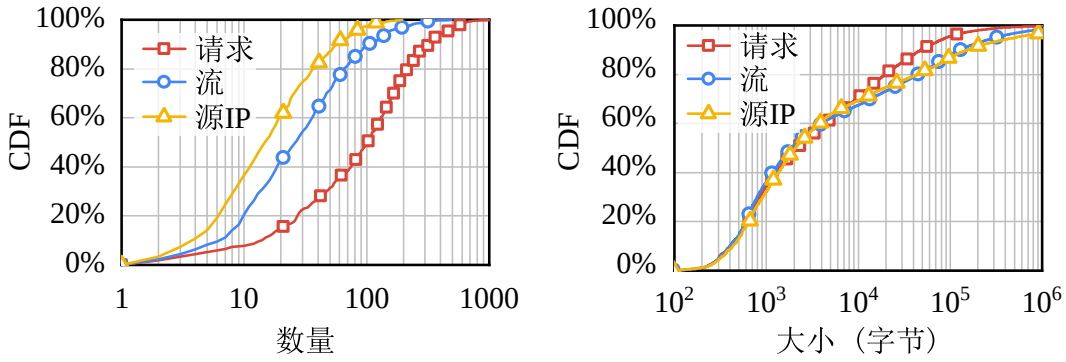
在继续介绍之前，我们考虑一下设置的问题。Confucius 是为住宅和终端用户接入点（例如 WiFi 接入点或蜂窝基站）部署的，我们的实验和数据涉及在这些设置中的应用程序使用，其中已知拥塞是常见的[5,250-251]。网络社区中有一个开放的讨论，探索拥塞在其他设置中的影响（例如在核心网[252]或数据中心[84]），但这些其他设置超出了 Confucius 的范围。此外，边缘路由器上的计算能力还使我们能够对流进行细粒度的流量管理，如 §8.7.5中所示。

8.2　动机

我们通过描述近来的趋势来解释为什么我们需要重新考虑队列管理。接下来，我们通过一个直观的示例来解释为什么现有的方法在队列管理和调度中都无法解决这些挑战。

（1）HRT 应用的兴起给队列管理带来了新挑战。虽然互联网始终携带多个应用程序，但是，繁荣的实时通信应用程序（例如视频会议，云游戏，虚拟现实）的出现，使得共享瓶颈链路变得特别具有挑战性。HRT 应用程序不仅需要低延迟，而且还需要在发送非常高的比特率时保持一致的低延迟[5,177]。尽管最近的无线技术（如 5G 和 WiFi 6[5,16,97]）取得了进展，但 HRT 对延迟的一致性要求经常被违反，从而带来糟糕的用户体验。为了实现 HRT 一致性目标，队列管理方案需要从保障**公平性**转向防止性能**波动**。

（2）在互联网中，波动是非常难以避免的。虽然直观上，通过队列管理方案来提供一致的性能，可以解决这个问题，但是两个关键特征使这项任务特别具有挑战性。首先，互联网流量通常是突发的。作为直觉，简单的页面加载会从多个源生成响应的突发。事实上，加载网页的中位数是 27 个流，而对于最多的 25% 的网站，该数字为 56 个流。例如，我们在图8.1(a)中展示了加载 Alexa 前 1000 网站所需的 HTTP 请求，同时发生的流（由 5 元组定义）和源 IP 的数量（测量时间：2022 年 7 月）。我们在一个测量点用 Chrome 测量并记录 HAR 日志[253]。其次，尽管大多



(a) 单个网页中的源 IP、流、请求的分布　　(b) 大小分布

图 8.1　加载 Alexa 前 1000 网站过程中需要的 TCP 流数量和大小

数队列管理方案都是针对基于丢包的拥塞控制算法设计的，但是今天的应用程序会根据其不同的目标运行多个不同的拥塞控制算法。事实上，顶级 Alexa 网站使用了 10 种不同的算法[254]。

研究问题。结合这些趋势，我们提出了一个问题：现在的网络队列机制（即主动队列管理和调度）能够**公平地和一致地**满足共享瓶颈链路的异构目标，同时又是实用的吗？

8.2.1　动机的示例

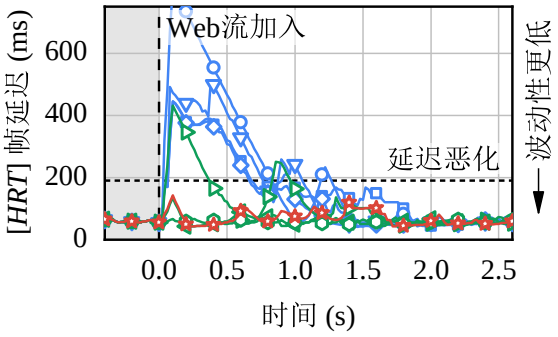


图 8.2　实时多媒体流延迟随时间变化情况

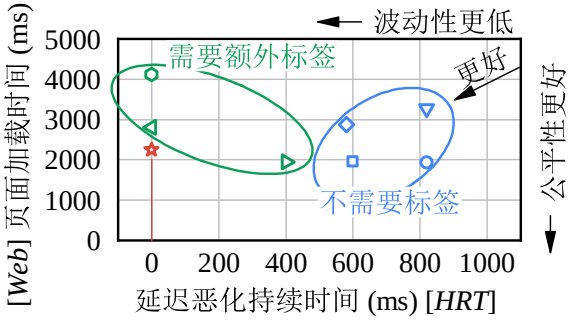


图 8.3　实时多媒体流和网页流的性能恶化

为了回答这个问题，我们做了一个直观的实验。假设用户正在进行视频通话，因此通过路由器传输了一个 HRT（*heavy, real-time*）实时音视频流。在 t=0s 时，另一个用户打开了一个网页并在同一瓶颈链路上创建了一批新的短流。这两个应用程序使用不同的拥塞控制算法以实现他们的目标。具体来说，实时多媒体流使用一种用于视频传输的[59]低延迟拥塞控制算法 Copa[49]，而网页使用 Cubic[56]。图8.2-8.4展示了当瓶颈链路由各种队列管理机制控制时，两个应用程序性能平均值以及公平性的结果随时间的变化。图8.2是一条预先存在的实时多媒体流（例如视频会议）与 Web 页面加载流（例如 `amazon.com`）竞争。类无关（蓝色）方案会导

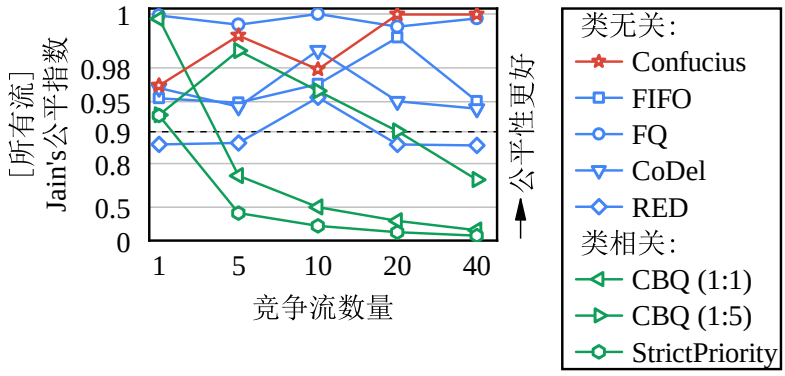


图 8.4　工作负载变化时的 Jain's 公平指数 (JFI)

致实时多媒体流的延迟不稳定地下降，而类有关（绿色）方案会导致 Web 流量的页面加载时间过长。图8.3展示了每种方案都会在实时多媒体流（延迟恶化持续时间）和网页流量（页面加载时间）之间寻求平衡。图8.4则展示了类有关的解决方案（例如 CBQ）对工作负载变化非常敏感。例如，CBQ 在不同的权重（1:1 或 1:5）下会在某些工作负载中导致公平性（JFI<0.9）较差。Y 轴未进行线性缩放。我们在 §8.7.2中更详细地解释了实验设置。虽然简单，但我们的示例实际上证明了公平性和波动之间的紧张关系。因此，我们从这个例子中得出的观察结果可以推广到其他流量混合和场景，如 §8.7中所示。

我们将现有机制分为两类：**类相关**和**类无关**。前者需要主机标记数据包以区分应用程序（例如视频会议或 Web）。后者不需要或不利用主机标签。

不幸的是，类相关机制无法在真实的设定下充分解决公平性和波动之间的紧张关系。具体来说，这些现有方案分别有下述一个或多个问题：

（1）性能波动：实时多媒体流在 Web 流加入时会遭受延迟恶化。类无关的方案，如 FIFO，FQ，RED 无法避免性能波动，从而伤害实时多媒体流。正如我们在图8.2中观察到的，当类无关方案（蓝色）管理瓶颈链路时，实时多媒体流体验高延迟。具体而言，HRT 的延迟增加了 4×，达到 400-800 ms。从长远来看，视频帧的端到端延迟超过 190 ms（图8.2中的虚线）会导致视频流停顿[245]。图8.5(a)和8.5(b)直观地解释了为什么简单的类无关方案，如 FQ 和 FIFO，如此不利于避免波动。图中，红色虚线标记了实时多媒体流的公平份额。请注意，当 Web 流到达时，实时多媒体流可用带宽减少如此剧烈，以至于实时多媒体流无法适应。

未能提供一致的延迟直观上并不是传统的主动队列管理方案（如 CoDel 和 RED）期望达到的结果。这些方案实际上试图控制端到端延迟[68,79-81]。然而，传统的主动队列管理方案无法平衡异构流的性能，因为它们是为基于丢包的 CCAs 而设计的[255]，并且不能有效地向基于延迟的 CCAs（如 Copa）传达拥塞，这些

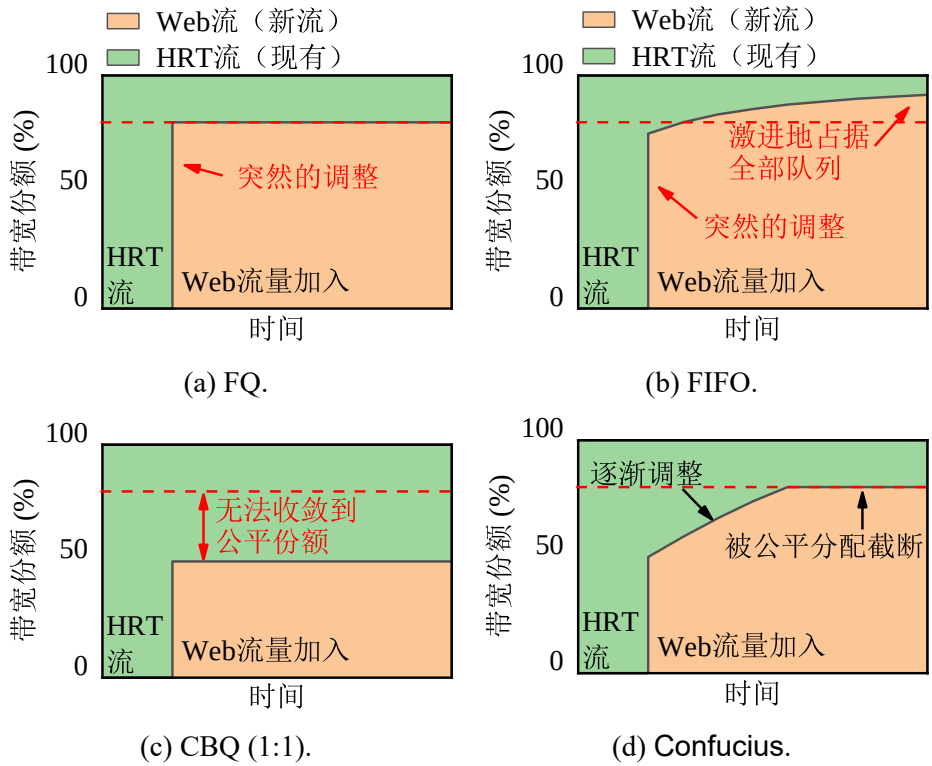


图 8.5　不同调度策略中有突发流量时带宽随时间变化的示意图

CCAs 是大多数实时流使用的[60]。对于多个延迟敏感的拥塞控制算法（包括 GCC 和 Copa），发送方不会将主动队列管理引起的丢包或 ECN 视为拥塞，因此在丢包率非常高之前不会降低其发送速率。因此，如图8.2和8.3所示，CoDel 和 RED 等主动队列管理导致了实时多媒体流的显着延迟退化。

（2）不公平性：Web 流或实时多媒体流遭受极端性能退化。如图8.2所示，类似 CBQ 这样的类相关方案，将数据包分成可配置服务速率或严格优先级的类相关队列，可以保护实时多媒体流。这是因为他们当且仅当在高优先级为空时才出队低优先级的数据包。然而，类相关方案也会导致不公平的分配（如图8.3所示），因为它们对 Web 流的惩罚过重（甚至饿死），会导致 Web 流的高页面加载时间。虽然理论上，CBQ 可以配置为公平的份额，但这需要对非常短的时间间隔内的确切工作负载（流之间的类比率）进行认知，这在实践中是不可行的。例如，我们测量了不同调度程序在更改 Web 流数量时可以提供的公平性，如图8.3所示。改变 CBQ 的类比率，可以使 CBQ 在 Web 流数量较少时保持公平，但是当 Web 流数量增加时，CBQ 会变得不公平。例如，CBQ（1：1）在只有两个流竞争时保持公平，CBQ（1：5）在有 6 个流竞争时保持公平，但在流的数量发生变化时，他们都不再公平。

（3）不实用性：要求端点正确标记其流量在互联网中是不现实的。除了对配置的敏感性之外，类相关方案还要求端点根据其重要性或目标对流进行标记，并根

据这些标记对流进行优先级排序。对于家庭路由器，这样的标签驱动管理是不现实的，原因如下。首先，标签会带来巨大的协调开销。实际上，用户需要根据其应用目标来使用标签，同时也需要与路由器协商这些标签的含义。其次，标签驱动管理假设端点是可信的和无错误的。实际上，发送方有动机将其流标记为更高的优先级。因此，这些方案主要适用于数据中心，其中端点和路由器都受到同一实体的控制（例如 LSTF[256]，pFabric[85]）。

尽管听上去简单，但这个例子教会我们两个关于如何处理各种目标或拥塞控制算法的两点内容：

收获 *1.* *在流量突发到达时立即实施带宽公平性会损害现有流的性能。这是因为拥塞控制算法的发送速率与瓶颈链路中可用带宽之间的差异很大。拥塞控制算法可能没有足够的信息来及时适应瓶颈链路中的带宽剧烈下降。*

收获 *2.* *由于不同拥塞控制算法或具有不同目标的流的拥塞感知不同，因此应该避免共享相同的队列。因此，即使是先进的主动队列管理方案也无法在正确的时间以正确的方式为每个独立的拥塞控制算法传递拥塞信息。*

8.3　主动队列管理设计

在本节中，我们解释了 Confucius 如何利用 §8.2中的收获来推动公平性和波动性之间的帕累托边界。为此，我们解释了 Confucius 如何在新流到达时重新分配带宽以避免性能波动。然后，我们解释了 Confucius 如何将带宽分配给新流和现有流（旧流）以实现公平的性能（公平性）。

8.3.1　通过谨慎的带宽重分配来避免性能波动

为了解决性能波动的问题，我们的第一个收获是，当流量突发到达时，立即实施公平性会损害现有流的性能，因为拥塞控制算法的发送速率与瓶颈链路中可用带宽之间的差异很大。然而，如果我们逐渐地、谨慎地控制流量的带宽分配，我们可以消除旧流的 CCA 发送速率与瓶颈链路上实际服务速率之间的差异，从而抑制波动。

为了理解为什么逐渐控制实时多媒体流的带宽分配比直接将其可用带宽减少到其公平份额会有优势，我们测量了严重延迟降级的持续时间 y。具体来说，y 表示实时多媒体流在延迟大于 190 ms①的时间间隔。我们将 y 作为 *Available Bandwidth Reduction Factor*（ABRF）的函数绘制在不同 CCA 的图8.6中。我们发现，CCA 对

① ITU 建议的网络延迟[245]

突然的大幅度带宽减少的响应非常差。例如，将 GCC 的可用带宽减少到其初始值的十六分之一（即 $ABRF = 16$）会导致 $y > 10$ 秒的视频帧停顿。有趣的是，我们发现，图8.6中的曲线 $y = f_{CCA}(ABRF)$，即我们表示 ABRF 与 y 之间的关系的曲线，随着 ABRF 的增加而超线性地增加。

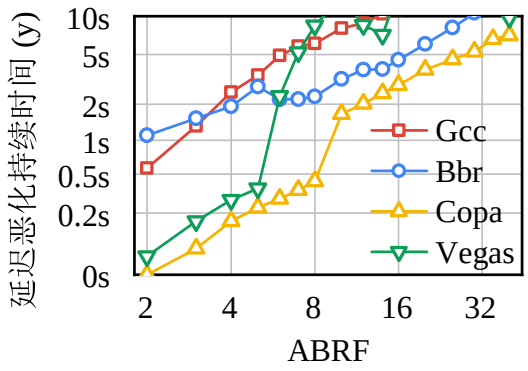


图 8.6　延迟恶化持续时间随可用带宽减小系数变化的测量结果

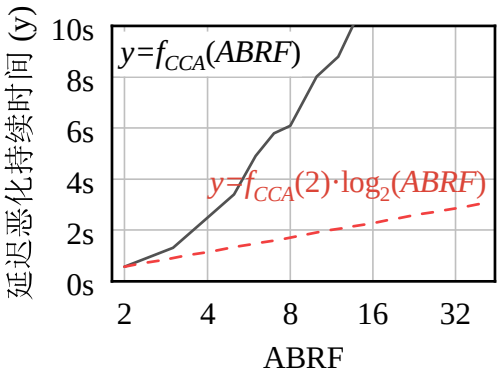


图 8.7　谨慎减少可用带宽有助于减少延迟持续时间的示意图

为了避免这种延迟降级，Confucius 逐渐减少实时多媒体流的可用带宽。例如，为了实现 ABRF 为 16 的最终值，可以平滑掉权重的改变而逐渐转为 $log_2(16) = 4$ 次带宽减少。这种指数（平滑）的带宽分配变化可以通过使用 EWMA 并在每次迭代中将实时多媒体流的带宽减半来实现。这将为 CCA 提供一个机会，通过其通常的拥塞信号来了解减少的带宽分配，同时在每次迭代中逐渐减少实时多媒体流的发送速率与可用带宽之间的差异，从而抑制波动。图8.7演示了理想情况下，这种方法的价值：相比于线性增长，延迟持续时间只会随着 ABRF 的增加而对数增加（由 $f_{CCA}(2)$ 调制，一个小常数）。

通过将一个对数衰减因子应用于实时多媒体流的可用带宽（而不是瞬时减少），Confucius 不再保留严格公平。直观地说，这可能会对短流造成严重的损害。然而，我们在 §8.4.2中证明，Confucius 保证短流的 FCT 总是在一个常数、可加因子的范围内，这个因子是严格公平分配下的 FCT。

8.3.2　平等地处理竞争流

解释了 Confucius 如何平等地处理实时多媒体流之后，我们现在来看看 Confucius 事实上如何在竞争流之间分配带宽。在高层次上，Confucius 首先将流分配到队列中，并为每个队列分配一部分可用带宽，如图8.8所示。w_i 表示 DWRR 调度中队列 i 的权重。

遵循收获 2，将流分配到不同的队列是必要且具有挑战性的。如果将所有旧流都放在单个 FIFO 队列中，那么如果流使用异构的拥塞控制算法，实时多媒体流

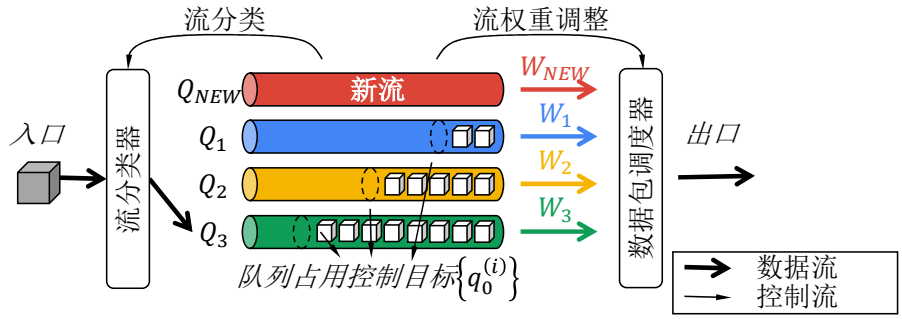


图 8.8　Confucius 设计概述

（例如 Copa）将会因为饥饿而被抑制[49]。但是，使用 FQ 将旧流分配到不同的队列可能无法为突发的旧流提供低延迟[257]。

Confucius 将流按照其队列占用量分配到队列中，按照具有相似性能目标的流不会互相伤害的原则。为了在系统中识别流的目标，Confucius 使用队列占用量。**我们发现，流会隐式地根据它们如何利用瓶颈队列来展示它们的偏好和目标**。例如，延迟敏感的应用程序将选择能够实现低延迟的 CCA，例如 Copa[49] 或 GCC[60]。这些 CCA 通过尝试保持瓶颈队列尽可能短来实现低延迟。相反，吞吐量优先的拥塞控制算法（例如 Cubic）将保持缓冲区充满以最大化吞吐量的利用率。这使我们能够根据队列占用量来识别流的延迟偏好：如果一个流的队列占用量很低，这意味着（i）该流试图不要过度利用队列；（ii）该流可以与具有类似行为的其他流共存。

通过将具有相似队列占用量的流分配到同一个队列中，具有不同队列占用量的流将不会相互影响。同时，由于要在固定数量的队列之间进行调度，因此延迟敏感的流（无论是否突发）都将具有一致的延迟。因此，Confucius 有一组队列，每个队列都设计用于容纳具有不同缓冲区占用量的旧流，以及专用于短流的队列。Confucius 采用 Deficit-Weighted Round-Robin（DWRR）算法在这些队列之间进行调度。当新流到达路由器时，Confucius 将其放入短流队列中。Confucius 将定期测量流特征并根据需要重新分类流。这样做可以使 Confucius 准确地测量流特征。为了进一步提高实践中性能的鲁棒性，我们引入了基于滞后的机制来重新分类流。我们在 §8.4.2中详细介绍了这种机制。

将流按照目标分类的一个自然问题是 *(i)* 如何在这些类别之间分配带宽；以及 *(ii)* 如何等待更改带宽分配的时间。对于前者，我们的见解是带宽分配需要取决于旧流和新流之间的数量比例。对于后者，我们的见解是避免过快地从旧流中将带宽移动到新流，以便旧流的拥塞控制算法有时间做出反应。

实际上，对于每个拥塞控制来说，遵循其反应时间意味着我们需要在各种拥塞

控制算法中调整 Confucius 的设计。为此，我们绘制了不同拥塞控制的响应曲线，并发现在带宽变化期间拥塞控制算法的反应时间总是超过某个阈值，Confucius 总是从温和的调整中受益。因此，我们可以为具有不同拥塞控制的流设计统一的权重调整算法。实际上，Confucius 有效地将注意力集中在最不活跃的拥塞控制上，以确保所有拥塞控制都有足够的时间做出反应。

8.4　基于时间的流权重调整

本节中，我们将介绍 Confucius 的权重调整机制（§8.4.1）。然后，我们分析了这种机制可以保证现有实时多媒体流和新到达的鼠流的有界性能损失（§8.4.2）。

8.4.1　调整机制

回忆一下，Confucius 将流分配到不同的队列中，并使用 DWRR 在这些队列之间进行调度。为了分配权重（即服务率），Confucius 使用以下过程。对于每个流，f，Confucius 首先计算一个权重，w_f；然后，对于给定的队列，Q，权重通过对 Q 中的所有流的权重求和来计算：

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

Confucius 设计的关键成分是计算每个流的权重。为此，Confucius 区分**新流**和**旧流**。事实上，Confucius 将新流分组到一个单独的队列中，称为 Q_{new}（如图8.8所示）。所有映射到 Q_1，Q_2，..，Q_n 的旧流都被分配一个流权重 $w_f = 1$，并由集合 $\mathcal{F}_{old}$ 统一指定。当新流到达时，首先将其映射到 Q_{new} 中，然后重新计算 Q_{new} 中所有流的权重，如下所示：

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

公式8.2的设计中有如下诸多考量：

基于时间的调整 $\left(2^{\lambda t}\right)$。如 §8.3.1所述，Confucius 会**逐渐减少**可用于实时多媒体流的带宽。为此，Confucius 会**逐渐增加**竞争新流的权重。这里，t 表示新流的年龄（以毫秒为单位），λ 是一个参数，用于控制流权重的调整速率——流权重每 $\frac{1}{\lambda}$ 毫秒加倍一次。λ 越高，新流收敛到其公平带宽份额的速度越快，实时多媒体流的可用带宽的减少也越突然。我们将在 §8.4.2中讨论 λ 如何影响性能损失的定量分析。

初始权重 $\left(\frac{|\mathcal{F}_{old}|}{|Q_{new}|}\right)$. 如果新流的初始权重太小，即使指数增长因子也会导致这

些新流的收敛期很长。特别是，当已有的旧流很多时，很难让少量的新流抢占他们应得的带宽份额。因此，我们将新流的初始权重与当前路由器中的**旧流数**相乘。对于每个新流，我们将初始权重设置为 $\frac{|\mathcal{F}_{old}|}{|Q_{new}|}$，其中 $|\mathcal{F}_{old}|$ 和 $|Q_{new}|$ 分别是旧流和新流的总数。初始权重的这种特定选择背后的道理是始终将旧流的带宽减少限制在比 2 减少的因子更温和的程度。在这种情况下，延迟降级的持续时间可以对数缩放，其底数为 $f_{CCA}(2)$，如图8.7所示。

上界 (min(..., 1)). Confucius 使用流权重阈值 1 来从 Q_{new} 队列中 “淘汰” 新流。一旦流的权重达到 1，该流就不再被视为新流，并根据流分类器的输出将其移动到其他队列中（§8.5）。

参数配置。Confucius 的 λ 参数的选择是设计的一个重要考虑因素。当 λ 较大时（例如，$\lambda \to \infty$），可用带宽的减少会导致波动性，而当 λ 较小时（例如，$\lambda \to 0$），新流会变得不公平（甚至饥饿）。此外，在设置此参数时，我们需要意识到不同的流，特别是具有不同拥塞控制算法的流，对相同的拥塞信号的响应不同（例如，Copa 需要 5 个 RTT 才能有效地降低其发送速率，而 BBR 的响应时间由其探测间隔 6-8 个 RTT 决定）。因此，我们希望将 λ 配置为尽可能快地降低可用带宽，同时受到底层拥塞控制算法的响应时间的限制。

为了解决网络上拥塞控制算法的异质性[254]，我们将 λ 设置为**最慢响应的延迟敏感 CCA 的探测周期的倒数**。这样，即使是最慢响应的拥塞控制算法也可以平滑地对带宽变化做出反应。根据图8.6中的实验，BBR 是最慢响应的拥塞控制算法，其探测周期为 6-8 个 RTT。因此，给定 Web 服务的典型 RTT 为 30-50 ms[258]，我们将 λ 设置为 0.004（ms^{-1}），以使得 $\frac{1}{\lambda}$ 的间隔为 250 ms。§8.7.2中的实验证明了不仅 BBR，而且其他几个拥塞控制算法都可以取得比较满意的结果。

8.4.2　理论分析

在本节中，我们将分析 Confucius 是否能为 HRT 和 Web 流提供一致的性能。在实时多媒体流和新流的竞争场景中，我们将分析 Confucius 是否能够保证实时多媒体流的延迟降级的持续时间有界，同时为新流提供公平的 FCT。我们在表8.1中列出了我们将使用的符号。

场景概述。考虑一个单独的实时多媒体流在瓶颈链路上运行。在 $t = 0$ 时，来了 N 个新流（例如，Web 流），每个流大小为 B，加入同一瓶颈链路并与现有流共享缓冲区。我们分析了现有流和新流的性能下降的情况。

CCA 模型。我们采用了一个简化的延迟收敛的拥塞控制算法模型[91,260]，其

表 8.1　本小节中用到的符号

参数和变量:	
B	每个新网页流的大小.
N	新网页流的数量.
k	拥塞控制算法的响应性.
q_0	一个拥塞控制算法尝试达到的延迟目标.
C	链路容量.
τ	拥塞控制算法的反馈回路 (一般是一个 RTT).
B_0	一条新流的初始突发 (例如，初始拥塞窗口[259]).
P	The scheduling policy.
函数:	
$s(t)$	实时音视频流在时刻 t 的发送速率.
$r(t)$	实时音视频流在时刻 t 的可用带宽.
$p(t)$	实时音视频流在时刻 t 的队列中数据包数量.
$q(t)$	实时音视频流在时刻 t 的排队延迟.

中延迟敏感的拥塞控制算法具有目标排队延迟 q_0。拥塞控制算法试图维持其排队延迟在此目标附近，通过增加或减少其发送速率来调整与目标的差异：

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

此处，$s(t)$ 是流的瞬时发送速率，$q(t)$ 是它经历的瞬时排队延迟，τ 是拥塞控制算法的控制回路。最后，k 是拥塞控制算法的响应性系数。我们讨论了不同拥塞控制算法的 k 如何变化。

延迟模型。接下来，我们分析队列中的包数 $p(t)$，在时间 t。在任何 $t > 0$，该数量满足以下关系：

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

其中，$p(0) = q_0 \cdot C$ 是缓冲区在稳态下的占用量，C 是链路容量。如果 $r(t)$ 表示实时多媒体流在时间 t 的瞬时服务速率（即可用带宽），则队列延迟可以写为：

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

这里面有两个我们关注的指标。第一个是实时多媒体流在给定调度策略 P 下

的**最大排队延迟**，q_P^{max}：

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

在这里，我们发现 q_P^{max} 可以作为延迟降级的持续时间的一个很好的代表，因为它为之前排队的包提供了一个**下界**，这些包从瓶颈队列中排出。

第二个指标是新流的 **FCT**，T，它可以表示为：

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

因为 FQ 提供了最公平的带宽分配（代表了公平性与非易失性之间的一个极端），所以我们使用 FQ 下 Web 流的流完成时间，T_{FQ}，作为我们的基线。然后，我们计算 $T_P - T_{FQ}$，表示策略 P 相对于 FQ 降低 Web 流性能的程度。

建立了这两个指标（**最大排队延迟**和**与 FQ 相比 FCT 的恶化**），我们评估了四种调度策略：FQ、FIFO、CBQ（1：1）和 Confucius。我们发现这些策略的可用带宽满足以下关系：

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

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

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

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

其中，对于 FIFO，N 是新流的数量，B_0 是新流的初始突发大小（例如，TCP 中的初始拥塞窗口）。然后，我们使用 Laplacian 变换来解决公式8.5中的性能降级 q_P^{max} 和 FCT 降低 $T_P - T_{FQ}$ 的微分方程。我们在表8.2中总结了**近似**的结果。在瞬态场景中，现有的调度策略要么具有发散的延迟降级，要么具有发散的流完成时间降级。随着工作负载的变化（N 和 B），具有发散性质的项在表中被标记为红色。

表 8.2　不同调度器的近似最大延迟（q_P^{max}）和 FCT 降低（$T_P - T_{FQ}$）

策略 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	$\approx 6q_0 + 15\tau + \frac{8\lambda}{k} + \frac{(10q_0+15\tau)\lambda^2}{k}$	$\approx \frac{\log_2 e}{\lambda}$

对于 FQ 和 FIFO，我们观察到延迟降级的持续时间与新流的数量 N 成线性关

系，因此是无界的，其中 N 可以在某些 Web 页面中超过 100（图8.1(a)）。直观地说，随着流加入瓶颈链路的数量增加，实时多媒体流的可用带宽下降的越多，导致实时多媒体流的延迟下降的越严重，从而导致显著的波动性。

在 CBQ 的情况下，预标记实时多媒体流的权重可以使策略将其分配固定的带宽，从而导致延迟降级的有界性。然而，如果权重并不能精确地匹配每个队列中的流的数量，CBQ 将收敛到不公平的解决方案，从而导致老鼠流的 FCT 降低变得无界（§8.2）。

最终，Confucius 为**两组流**提供了有界的性能下降。一方面，Confucius 确保实时多媒体流的延迟下降是一个仅取决于 CCA 队列延迟目标（q_0）、CCA 的响应性（k）、CCA 的反馈循环的持续时间（τ）和衰减参数（λ）的常数[①]。另一方面，Confucius 还可以确保老鼠流的 FCT 下降是随着衰减参数（λ）的增加而增加的常数因子，随着流大小的增加而变得微不足道。

8.5　基于使用情况的流分类

如 §8.3.2中所述，Confucius 旨在将流分类到具有专用队列的组中，以便根据它们消耗缓冲区空间的积极程度进行分类。在本节中，我们首先介绍了我们在将流分类到不同队列时的设计考虑 (§8.5.1)。然后，我们介绍了基于滞后的机制来稳健地对流进行分类 (§8.5.2)。

8.5.1　设计考量

Confucius 将短流放到一个单独的队列 Q_{new} 中，并将具有不同缓冲区积极性的长流分类到不同的队列中。因此，我们需要设置一系列队列 $Q_1, Q_2, \cdots, Q_n$ 来容纳具有不同缓冲区积极性的流[②]。队列索引随着缓冲区目标而增加，即 Q_1 将比 Q_3 短，如图8.8所示。具体而言，我们将队列 Q_i 尝试维持的缓冲区占用量称为 $q_0^{(i)}$。实现这一点带来了两个问题。首先，我们应该为路由器设置多少个队列来容纳异构流。其次，如何将流的缓冲区占用与队列 Q_i 尝试维持的目标 $q_0^{(i)}$ 匹配。我们将在以下部分回答这两个问题。

需要设置的队列数量。首先，我们需要确定为实例化 Confucius 而设置多少个队列。为此，我们需要估计互联网上普遍使用的有多少个不同队列行为的拥塞控制算法的类别。为此，我们测量了 7 个拥塞控制算法（互联网中使用的前 5 个拥塞控制算法[254]加上两个最近的延迟敏感拥塞控制算法，即 GCC 和 Copa），以及

① 在实践中，当使用 40ms 的 RTT 时，表8.2中的近似上界 $q^{max}_{\text{Confucius}}$ 大约为 640ms。正如我们在 §8.7.2中实验性地展示的，使用 Confucius 的延迟下降要远低于这个值。

② 我们使用每个队列的缓冲区占用作为最大队列长度。

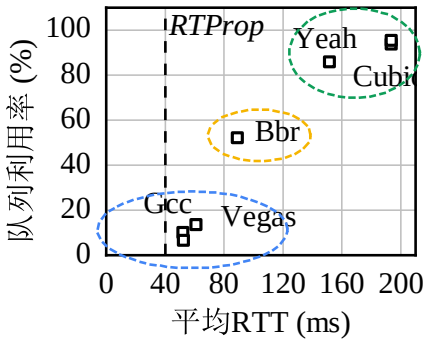


图 8.9　不同拥塞控制算法的队列利用率与延迟的关系

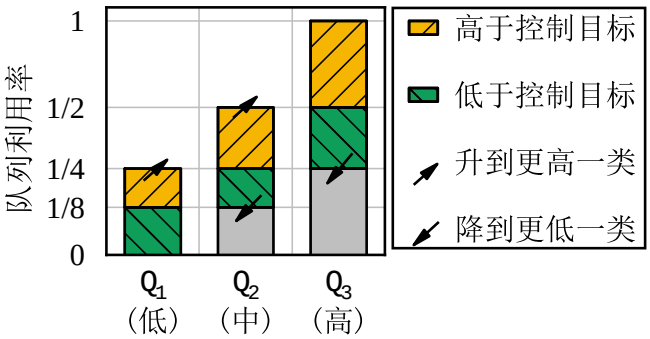


图 8.10　基于迟滞的流再分类机制

真实世界的带宽数据[5]。我们还测量了发送方的网络 RTT 和应用程序层性能（包括套接字缓冲区中的延迟和重传）。较低的 RTT 和应用程序延迟表明，这样的给定拥塞控制算法对延迟更敏感。如图8.9所示，GCC、Copa 和 Vegas 具有较低的网络 RTT 和应用程序延迟。因此，延迟敏感的应用程序可以选择这些拥塞控制算法以实现更低的延迟。Cubic、Yeah 和 Illinois 的延迟要高得多，而 BBR 则介于两者之间。我们观察到拥塞控制算法聚集在三个簇中（图8.9中的虚线圆）。实验使用来自 [5] 的真实 WiFi 带宽数据集进行模拟。具体而言，GCC、Copa 和 Vegas 的队列占用率小于 20%；Cubic、Illinois 和 Yeah 的队列占用率大于 80%；而 BBR 的队列占用率则介于两者之间。因此，我们设置了三个队列，并使用这三个簇中的平均队列占用率作为我们的目标 $\{q_0^{(i)}\}$。我们希望其他拥塞控制算法落入这三个代表性类别之一，如果不是，我们可以配置 Confucius 以支持更多的队列。

实际挑战。虽然可以离线地对流进行分类，如上所述，但 Confucius 无法在线使用相同的方法。事实上，Confucius 以线速运行，而流不会预先标记其拥塞控制算法。因此，实际上在线推断流的缓冲区积极性是具有挑战性的。首先，流的缓冲区积极性可能需要很长时间才能表现出来。例如，Confucius 将无法对仅持续几个 RTT 的短流进行分类（§8.2）。其次，网络条件也会影响测量，从而欺骗 Confucius。例如，可用带宽的下降会导致缓冲区占用量的增加[5]，这并不一定意味着流积极占用缓冲区。最后，流的缓冲区积极性可能会随时间而变化。例如，受限/拥塞在其他地方（在不同路由器上）的 Cubic 流将不会被 Confucius 分类为激进使用队列的流。因此，这样的 Cubic 流可以与其他延迟敏感流共享队列。但是，当瓶颈移动到当前路由器时，这个 Cubic 流将变得激进地占用缓冲区。因此，我们需要定期监视每个流在其当前队列中占用的缓冲区占用量，并重新考虑其分类。我们在下一节详细介绍我们的算法。

8.5.2　基于迟滞的调整

为了允许重新分类，同时避免流的分类发生振荡，我们引入了迟滞机制。整体分类步骤如下：

新流的重新分类。对于新流队列 $\mathcal{F}_{new}$ 中的流 f，当流准备好（其权重达到 1）从新流队列 Q_{new} 移动到一个旧队列时（我们在 §8.4.1中详细介绍），我们测量该流的缓冲区占用量 q_f，即该队列中属于流 f 的数据包的数量。然后，我们找到最接近的 Q_i 以容纳该流的队列 i。

周期性调整。 Confucius 定期检查流和队列，并根据需要移动流。

1. **队列间检查。** Confucius 检查每个队列 Q_i，并比较其当前占用率（$\frac{\sum_{f\in Q_i} q_f}{\sum_{g\in Q_i} q_g}$）与其目标占用率（$q_i^{(i)}$）。如果当前占用率大于目标占用率，即：

$$\frac{q_f}{\sum_{f\in Q_i} q_f} \geqslant \frac{1}{|Q_i|} + \alpha \tag{8.9}$$

其中，$\alpha > 0$ 是一个迟滞系数，表示流在当前队列中流动过于积极。Confucius 将从队列 Q_i 中提升该流到队列 Q_{i+1}，以使 Q_i 接近其控制目标。同样，如果流的缓冲区占用量显著地低于其在队列中的公平份额，即：

$$\frac{q_f}{\sum_{f\in Q_i} q_f} \leqslant \frac{1}{|Q_i|} - \alpha \tag{8.10}$$

那么，Confucius 将该流从队列 Q_i 降级到队列 Q_{i-1}。基于我们在图8.9中的观察结果，我们将 α 设置为 10%。我们在 §8.7中的评估表明，Confucius 的性能对工作负载和拥塞控制算法不敏感。

2. **队列级检查。**Confucius 检查每个队列 Q_i，并检查其长度是否在目标范围内。如果队列的长度超过两个邻居队列的控制目标之间的安全区域，Confucius 将当前队列中的所有流移动到更高或更低的队列，如图8.10所示。注意，只有当队列占用情况严重偏离当前类时，才将其移动到其他类。这是必要的，因为队列内检查仅关注跨流相对占用。因此，它无法识别流在当前队列中积极但比当前队列的控制目标更积极的情况。例如，假设有两个受限/拥塞在其他地方（在不同路由器上）的 Cubic 流，这些 Cubic 流之前被分类为 Q_1（最不积极）。当这些 Cubic 流开始在缓冲区占用方面变得积极时，Confucius 将需要将它们移动到不同的队列，以保护可能加入的延迟敏感流。

尽管这些操作似乎很复杂，但这些操作都在 Linux 基于边缘路由器的能力范围内。事实上，我们在 §8.6中实现了一个完整的原型。

8.6　系统实现

在 Linux 内核中实现 Confucius 有一些挑战。我们在下面讨论这些挑战以及我们的解决方案。

8.6.1　重分类中的保序性

流可以在运行时移动到另一个类中。因此，我们需要确保在重新分类 Confucius 的某个流时保持顺序。为此，我们在 Confucius 中采用虚拟类设计。在新数据包的入队过程中，我们将 `sk_buff` 绑定到每个流。在出队过程中，我们搜索所有绑定到确定类的流，并出队具有最早入队时间的数据包。这样，当将流移动到另一个类时，我们只需从前一个类重新绑定流的指针到新类。

8.6.2　降低计算开销

为了在 Linux 内核中实现 Confucius 并优化执行开销，我们需要严格优化计算开销。具体来说，我们有以下两种实现：

(1) 指数运算的位移操作。 Confucius 根据流的年龄对流进行重新加权，但内核中的浮点数计算是昂贵的。因此，我们将新流的权重量化为 $\frac{1}{128}$ 的单位。我们遵循 EWMA 的实现，并使用位移操作来进行指数变化的权重，即每 $\frac{1}{\lambda}$ 毫秒将权重左移一位。

(2) 重分类和重新加权的周期性。重分类和重新加权不一定需要针对每个数据包进行。对于重新加权，如我们之前讨论的，我们只需要针对某个流每 $\frac{1}{\lambda}$ 毫秒重新加权一次。当我们令 $\lambda = 0.004$ 时，这意味着每 250 毫秒重新加权一次。对于重新分类，我们至少需要在将一个流移动到新类中观察一段时间的结果，以测量队列的利用率，这至少应该是一个 RTT，以充分观察发送者在新类中的行为。因此，我们也以周期性的方式重新分类流 – 我们将重新分类周期设置为 100 毫秒。

8.7　实验评估

我们首先介绍我们的实验设置（§8.7.1）；然后我们通过回答以下问题来评估 Confucius：

- Confucius 如何在不同的拥塞控制算法、真实世界 Web 数据集上进行公平和稳定性的权衡？Confucius 在不同的拥塞控制算法上保护实时多媒体流不受延迟降级的影响，同时加载 95% 的网站。相比之下，使用类无关方案（如 FQ 或 FIFO）时，该百分比低于 30%（§8.7.2）。

- Confucius 对负载变化有多敏感？我们变化了流的大小和数量，发现 Confucius 始终保持良好的性能（以实时多媒体流的延迟降级和 Web 流的 PLT 降级为衡量标准），始终遵循我们的理论分析（§8.7.3）。
- Confucius 如何在不同流有不同拥塞控制算法的情况下扩展？我们在不同拥塞控制算法的流的共存下测试 Confucius，并证明 Confucius 可以正确地根据流的行为将流分离，并为所有流提供一致的性能（§8.7.4）。
- Confucius 在实验床原型中表现如何？我们将 Confucius 集成到 Linux 内核 4.4.0 中的 `qdisc` 模块，并使用真实的 HTTP 请求跟踪评估 Confucius。Confucius 可以将延迟降级的持续时间减少 60% 以上，同时开销合理（§8.7.5）。
- Confucius 在不同设置下的表现如何？我们展示了 Confucius 在多个实时多媒体流竞争、带宽探测拥塞控制算法和不同瓶颈下仍然能够优于基线（§8.7.6）。

8.7.1　实验设置

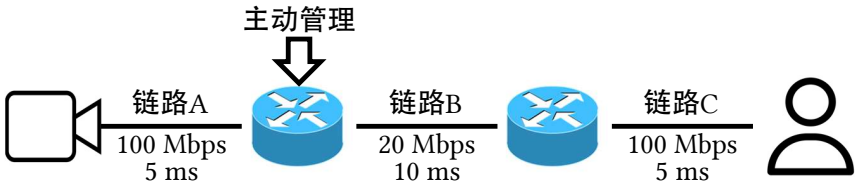


图 8.11　实验设置。

ns-3 设置。在 §8.7.2-8.7.4中，我们使用 `ns-3.34` 评估 Confucius 的性能。我们设置了一个线性拓扑，并将瓶颈链路的容量限制为 20Mbps，这是 [5] 中 WiFi 数据集的平均带宽，如图8.11所示。总的来说，基于 [5] 的测量结果，我们将往返传播延迟设置为 40ms。我们在 §8.7.6中进一步改变 RTT 和瓶颈。我们在 `ns-3` 中采用视频会议应用程序，其中流是实时多媒体流。我们为实时多媒体流采用不同的延迟敏感拥塞控制算法，包括 Copa[49]、GCC[60] 和 BBR[57]。Web 流使用 Linux 内核中的默认拥塞控制算法–Cubic[56]。

Linux 内核设置。在 §8.7.5中，我们将 Confucius 作为 Linux 内核 4.4.0 中的队列管理器（`qdisc`）模块实现，并在具有 Intel Xeon E5-2620 v4 CPU 的机器上评估 Confucius 的性能。我们使用 Copa[261] 的官方 CCP 实现。

Web 数据集。为了构建一个真实且相关的 Web 流量数据集，我们采取了两个步骤。首先，我们收集了 Alexa Top-1000 网站（2022 年 7 月，分布情况见图8.1）。其次，我们加载了这些网站并测量了它们触发的 HTTP 请求的大小。有了这个数据集，我们就可以重放这 1000 个网站的跟踪来测试各种场景。我们计划发布我们

的数据集。

基线。我们比较了 Confucius 的多种调度和主动队列管理的基线。对于这些基线中的参数，我们使用 Linux 内核 4.4.0 或 `ns-3.34` 中的默认参数。

(1) `FIFO` 和 (2) `FQ`，最常用的调度器。

(3) `SJF`（最短作业优先）优先级高于长流。由于我们无法知道哪个作业更短，我们用作业的年龄（即 PIAS[84]）来近似作业的长度，即总是优先级较高的流，这与 Confucius 正好相反。

(4) `HHF`[262]（Heavy-hitter filter）将小流和大流区分开来，为每个类别分配固定的带宽。

(5) `CoDel`[68] 和 (6) `RED`[79] 将在队列溢出之前丢弃数据包，以通知发送者拥塞。

(7) `CBQ` 将来自不同应用程序的流根据其标签放入不同的类中。我们将两个类的权重设置为 1:1 和 1:5，并分别评估性能。

(8) `StrictPriority` 如果将流标记为 HRT，则严格优先于实时多媒体流。

(9) `DualQ`[83] 是 L4S[246] 中最近提出的调度器，它使用标签保护延迟敏感流。

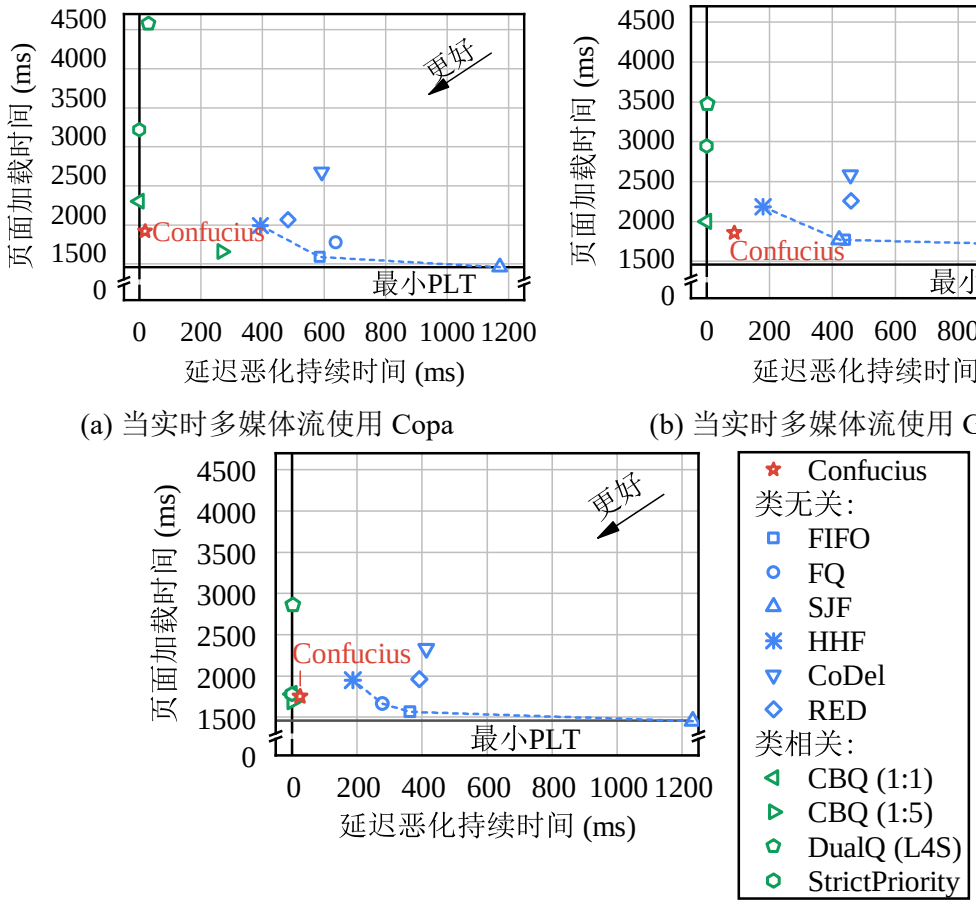
指标。在实验中，我们关注以下指标。

- **延迟恶化的持续时间**是视频帧的延迟大于 190ms 的持续时间。这直接反映了用户对视频卡顿的体验[5,245,263]。我们使用这个指标来评估波动性如何影响实时多媒体流的性能。
- **页面加载时间（PLT）**是网页中最后一个 HTTP 请求完成的时间。我们使用这个指标来评估 Web 流量的性能。PLT 恶化是指与 `FQ` 相比的延迟增加。

此外，我们还在不同的实验中评估其他指标，我们将在相应的实验中详细说明。

8.7.2　Confucius 真实负载实验

仿真场景。在 t=0 时，我们从视频会议应用程序开始实时多媒体流。在 t=10s 时，我们构建与 Alexa Top 网站之一相关的请求。所有流都是活动的，即我们不是重播预先记录的流量。我们运行相同的场景 1000 次，每个网站一次。在每次运行中，我们测量视频流的帧延迟大于 190ms 的持续时间（延迟恶化）。我们还测量了来自不同网站的网页加载时间。我们重复整个实验三次，每次考虑实时多媒体流使用不同的拥塞控制算法。我们在图8.12中汇总并呈现平均结果。虚线表示无类别基线的 Pareto 前沿。我们在不同子图中更改实时多媒体流使用的拥塞控制算法，并观察到 Confucius 在所有实验中都有类似的性能提升。实验同时得出如下观察与结论：



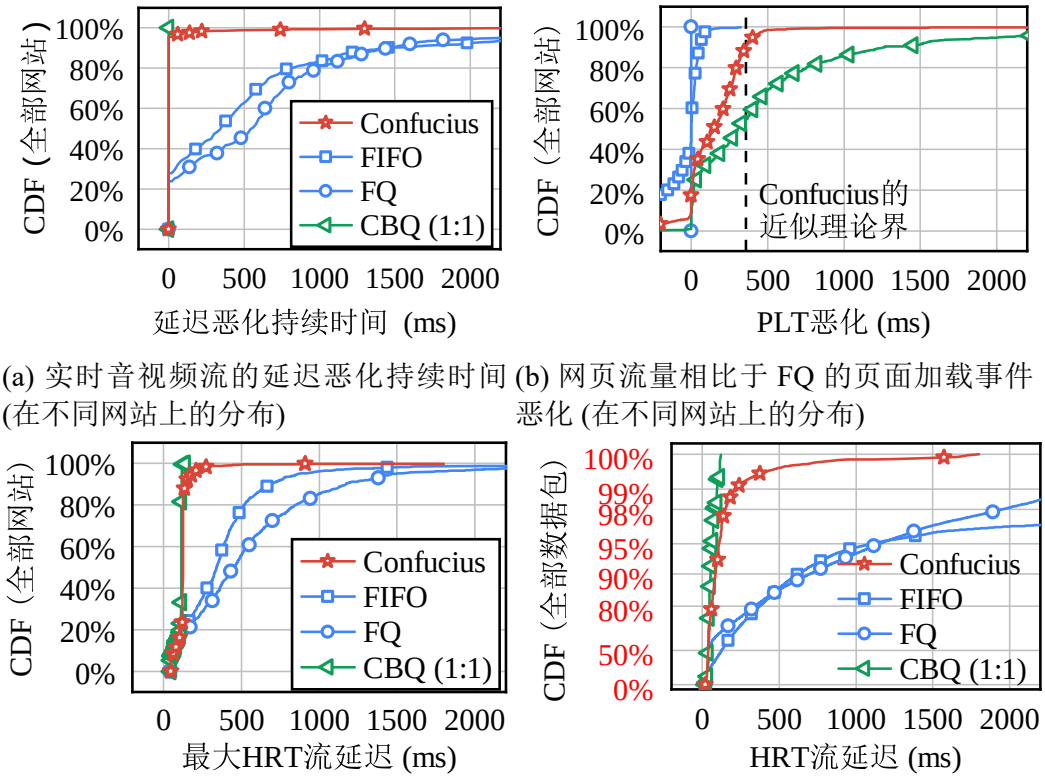
(a) 当实时多媒体流使用 Copa　(b) 当实时多媒体流使用 GCC

(c) 当实时多媒体流使用 BBR

图 8.12　实时多媒体流和网页流之间的性能权衡

（1）Confucius 在不同的拥塞控制算法上可以在视频与 Web 之间平衡性能。在图8.12中，我们观察到，不使用来自端点的标签的无类调度器（即标记为蓝色的调度器）会导致长时间的视频停顿。例如，当使用 `FQ` 和 `FIFO` 时，视频流平均会延迟 600ms。使用标签的类调度器（即标记为绿色的调度器）保护预标记的视频流，但会显着降低 Web 流量的 PLT。更糟糕的是，正如我们在 §8.2中讨论的那样，不可能假设端点将所有流正确标记。Confucius 不仅减少了与现有无类方案相比的延迟恶化的持续时间，而且几乎与类方案相当。此外，Confucius 在不同的拥塞控制算法之间保持了低 PLT 恶化。值得注意的是，Confucius 将类无关调度器的帕累托边界（即虚线蓝线）向前推进。当视频流使用其他拥塞控制算法（如 BBR 或 GCC）时，Confucius 的结果与图8.12类似。

（2）Confucius 能够在多于 95% 的网站上保护实时多媒体流，同时不牺牲其性能。我们进一步将图8.12(a)中的分布分解为图8.13。图8.13(a)显示了当视频流遇到来自数据集中不同网站的 Web 流时的延迟恶化持续时间的分布。使用 `FQ` 或 `FIFO`，实时多媒体流将在超过 70% 的网站上遇到延迟恶化（帧延迟 >190ms），其



(a) 实时音视频流的延迟恶化持续时间 (在不同网站上的分布)

(b) 网页流量相比于 FQ 的页面加载事件恶化 (在不同网站上的分布)

(c) 当网页流量到达时，实时音视频流的最大帧延迟 (在不同网站上的分布)

(d) 在网页流量竞争时，实时音视频流的逐包延迟分布 (在不同数据包上的分布，对数坐标)

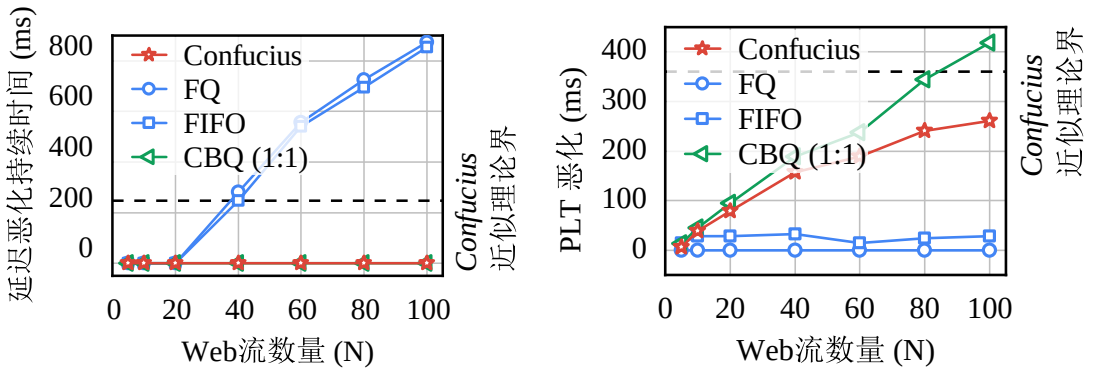
图 8.13　图8.12(a)中的结果分布

中一半甚至持续 520ms（在 `FIFO` 的情况下）和 660ms（在 `FQ` 的情况下）。相比之下，使用 Confucius，实时多媒体流在遇到来自超过 95% 的网站的 Web 流时不会遇到任何延迟恶化。重要的是，Confucius 不会过度惩罚 Web 流量——90% 的网站的 PLT 只会增加不到 360ms，如图8.13(b)所示，这几乎与我们之前的理论分析相符。我们还在图8.13(c)和8.13(d)中展示了遇到不同网站时实时多媒体流的最大延迟分布和所有数据包的延迟分布。这进一步证明了，Confucius 能够不仅在延迟恶化的持续时间上，而且直接在原始延迟上控制延迟波动。使用 GCC 和 BBR 的结果类似。

8.7.3　Confucius 在负载变化时的表现

在本小节中，我们在更实际的设置中测试了我们的理论分析。具体来说，我们研究了 Confucius 是否可以在不同的工作负载中提供一致的性能。为此，我们通过改变 Web 页面中的流数量和 Web 流的大小来改变工作负载。我们测量了不同场景下延迟恶化的持续时间和与 `FQ` 相比的 PLT 恶化。

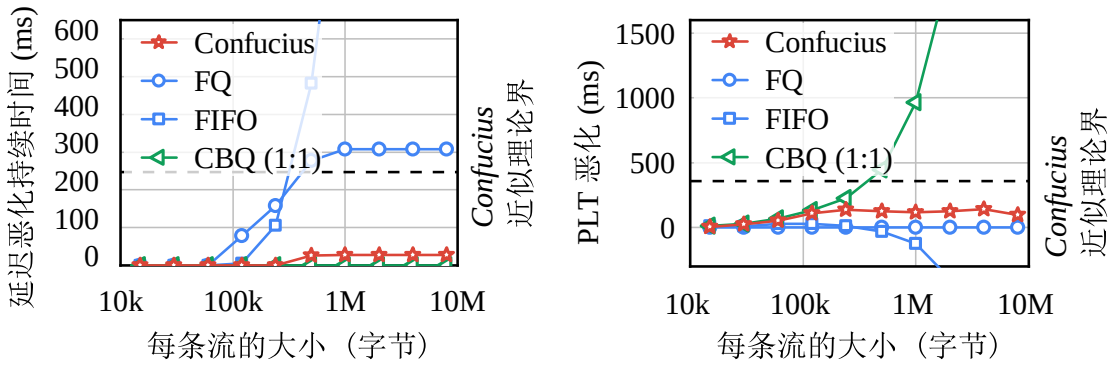
实验发现，Confucius 的延迟恶化是被一个理论上估计的阈值所限制的。这也



(a) 实时音视频流延迟恶化时间　　(b) 网页流量相比于 FQ 的页面加载时间恶化

图 8.14　在不同数量的 Web 流中，每个流的大小为 15KB 时的性能一致性

证实了本章的分析。我们在图8.14(a)中将 Web 页面中的流数量从 5 到 100 变化，每个流的大小为 15KB，并总结了我们的结果。`FQ` 和 `FIFO` 的延迟恶化随着流数量的增加而增加。例如，当 Web 流的数量达到 60 时，使用 `FQ` 或 `FIFO` 时，实时多媒体流的延迟会恶化超过半秒。相反，Confucius 在这种设置下保持零延迟恶化，与 `CBQ`（使用标签）类似。我们进一步比较了实验结果和我们在 §8.4.2中的分析结果。如图8.14中的黄色虚线所示，实验结果证实了表8.2中对 Confucius 性能的理论分析。



(a) 实时音视频流延迟恶化时间　　(b) 网页流量相比于 FQ 的页面加载时间恶化

图 8.15　在不同大小的 Web 流中，每个流的数量为 5 时的性能一致性。

我们进一步改变了 Web 流的大小（从短流到长流），并观察 Confucius 是否能够处理所有类型的竞争流量。我们将 Web 流的大小从 15KB 变化到 9MB，并运行 5 个具有相同大小的流来与实时多媒体流竞争。随着流大小的增加，竞争流从短流（例如，Web）变为长流（例如，FTP）。在这时，当使用 `FIFO` 时，由于无法提供流之间的交叉 CCA 公平性，实时多媒体流将遭遇延迟恶化，如图8.15(a)所示。实时多媒体流使用 FQ 也有数百毫秒的长延迟恶化。相比之下，Confucius 仍然能够在同一时间内实现实时多媒体流的延迟恶化时间为零，以及 Web 流的 PLT 恶化是有界的。

8.7.4　异质流分类

在本小节中，我们将 Confucius 的流分类机制放大，以研究其对延迟和公平性的影响。我们发现 Confucius 可以将具有相同 CCA 的流分组在一起，而无需任何先验知识，这在某种程度上提高了性能。

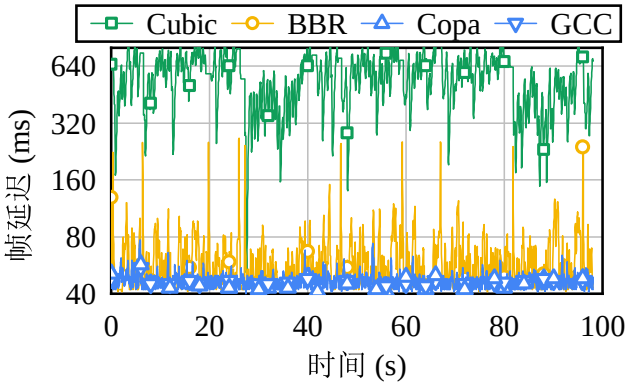
我们同时运行实时多媒体流的四个不同的 CCA：一个 Cubic 流，一个 BBR 流，一个 GCC 流，以及一个 Copa 流，持续 100 秒。我们在图8.16(a)中绘制了每个流的延迟随时间的变化。我们在图8.16(a)和8.16(b)中展示了使用 Confucius 时这些流的帧延迟和分类结果。在这个实验中，我们还测量了 JFI，以呈现使用不同方案时的公平性。如图8.16(c)所示，我们比较了 Confucius 和基线之间的公平性（JFI）和延迟敏感流（Copa 和 GCC）的延迟。我们还将相同的实验设置下的结果（Copa 和 GCC 流的延迟，以及所有流之间的 JFI）与其他调度程序进行了比较，如图8.16(c)所示。我们发现，使用 Confucius 时，Copa 和 GCC 流即使与 Cubic 和 BBR 共享瓶颈链路，也能够保持低的端到端延迟。同时，它们也可以获得合理的带宽公平性——在这个实验中，JFI 为 0.98，如图8.16(c)所示。

为了理解 Confucius 的优异性能，我们查看其随时间的分类，并验证 Confucius 在实践中如我们所期望的那样工作。我们做了两个观察。首先，Confucius 可以将具有不同 CCA 的流分类到不同的队列中。如图8.16(b)所示，Copa 和 GCC 流可以稳定地分类到低占用队列（Q_1，蓝色），BBR 流分类到中占用队列（Q_2，黄色），Cubic 流分类到高占用队列（Q_3，绿色）。这遵循我们在图8.9中的前一观察——Copa 和 GCC 都表现出类似的低缓冲区占用，而 Cubic 则积极占用缓冲区，BBR 在中间。这样，具有不同队列占用的流就可以相互隔离。此外，我们注意到 Cubic 流可以暂时与 BBR 流处于同一队列中，如图8.16(b)中绿色条中的黄色线所示。其实，这对 Confucius 是有益的，因为 Cubic 流在探测期间（在这种情况下，是在 Q_2 中）有时具有较低的队列占用。其次，具有不同 CCA 的流可以在具有相似缓冲区占用的情况下共存于同一队列中。在这个实验中，Copa 和 GCC 流可以放在同一队列中，因为它们具有相似的缓冲区占用。如图8.16(a)所示，这两个流在整个时间段内仍然具有一致的低延迟。

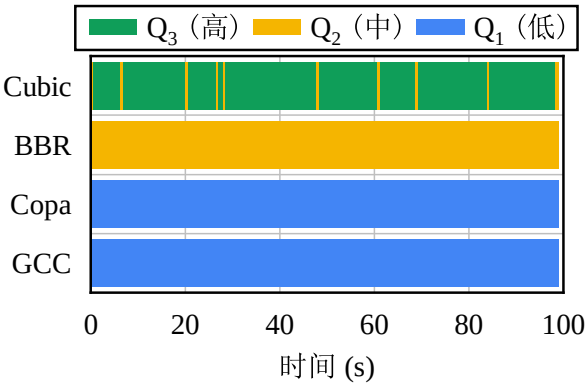
8.7.5　实验床实验

我们还在基于 Linux 内核的测试床上进行了性能评估。我们发现，Confucius 在基于 Linux 内核的实现中也能够实现显著的性能提升，同时只增加了很小的处理延迟。

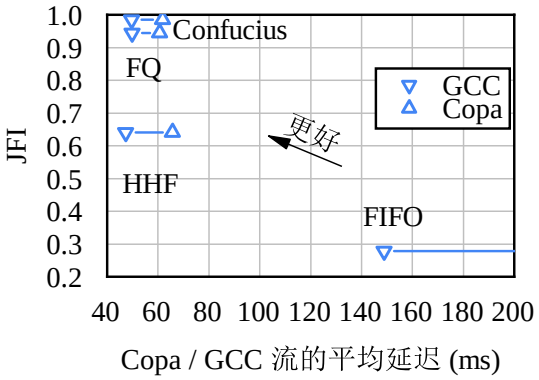
我们运行了一条 iperf3 流，将拥塞控制算法设置为 Copa，并测量 iperf3 报告



(a) 不同流的帧延迟变化情况。

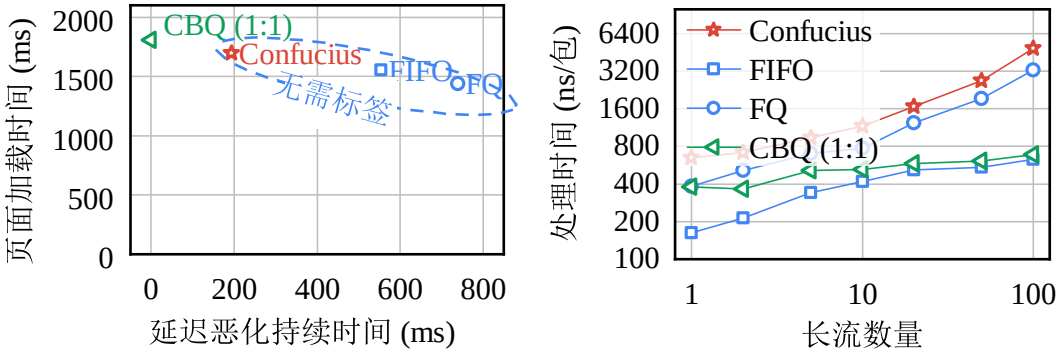


(b) 分类结果随时间变化的情况。



(c) 和基线相比的 JFI 和延迟.

图 8.16　四个不同的拥塞控制算法的流在同一个瓶颈路由器上运行的情况



(a) 实时音视频流的延迟恶化 vs. 网页的加载时间.

(b) 逐包处理延迟

图 8.17　在基于 Linux 内核的测试床上的结果

的延迟敏感流的延迟。然后，我们基于 Python 设置了一个 HTTP 服务器，以向客户端提供我们收集的 Web 跟踪。我们还测量了 Confucius 和基线的计算开销。我们在 Linux `tc` 中记录了入队和出队操作的处理时间，其中 Confucius 中的重新加权和重新分类都是在其中实现的。

如图8.17(a)所示，Confucius 可以显著减少延迟恶化的持续时间多达 60%，而不需要在每个数据包上标记延迟敏感流。这个结果与我们在图8.12(a)中的仿真结果类似。此外，86% 的网站在使用 Confucius 时不会出现延迟恶化。值得注意的是，这个数字在 FIFO 和 FQ 中仅为 56% 和 30%。

我们改变了长流的数量，以观察处理时间的变化。请注意，Confucius 对短流的数量不敏感，因为它们都属于新流队列。如图8.17(b)所示，Confucius 会略微增加每个数据包的处理时间，相比于 FQ。然而，即使有 100 个并发的长流在同一队列中，每个数据包的处理时间仍然是 5 μs，这表明每个数据包的处理速率为 200 kpps，或比特率为 100 Mbps~2.4 Gbps（取决于数据包大小）。请注意，Confucius 主要是为最后一跳路由器（如家庭路由器）而设计的。这可以满足家庭接入点或最后一跳路由器的日常使用。我们强调，Confucius 的内核实现可以在未来进一步优化以实现高性能执行。我们将在未来探索 Confucius 在多个流中的表现（例如在核心网络中的路由器）。

8.7.6　微基准

我们进一步评估了 Confucius 在一系列微基准设置中的性能。我们证明了 Confucius 的滞后机制（§8.5.2）可以与带宽探测型的拥塞控制算法（如 BBR）一起工作，并稳定地正确地分类流。我们进一步证明了，如果瓶颈不在部署 Confucius 的路由器上，Confucius 也不会产生任何副作用。最后，我们还证明了即使有多个实时多媒体流同时竞争，Confucius 仍然能够同时处理这些流，并且相对于基线提

供了显着的性能改进。这些结果都在 [264] 中。

8.8　本章小结

本章提出了 Confucius，一种新的旨在平衡公平性和波动性的队列管理方案。Confucius 通过观察流的缓冲区占用情况来对流进行分组，从而实现这一目标。Confucius 逐渐调整每个流的权重，并使用这些权重来设计每个队列的服务速率。这样做可以使 Confucius 减轻波动，从而改善实时多媒体流的性能。基于 Linux 内核的仿真和 ns-3 的仿真表明，Confucius 可以将导致视频流延迟恶化的网站数量从 70% 减少到 5%，同时只增加了很小的开销。

第 9 章　总结与展望

9.1　工作总结

实时多媒体传输是当前互联网上的重要应用。随着人们日益增长的美好生活需要，实时多媒体传输应用面临的延迟要求越来越高。针对实时多媒体传输的延迟优化有重大意义。现有的方案目前大多仍聚焦于中位数、90 分位等一般情况下的延迟，而忽视了对 99.9 分位甚至 99.99 分位的延迟的优化。在许多如云游戏、远程手术、虚拟现实等实时多媒体应用中，万分之一的卡顿都可能会带来严重的后果。相较而言，本工作的实时多媒体传输延迟优化更多聚焦于那些出现频率为千分之一、万分之一的卡顿事件。

不同于中位数、90 分位延迟等有明确的瓶颈（一般为传播延迟），当我们讨论出现概率为千分之一、万分之一的极致尾延迟时，任何一个组成部分的高延迟均有可能导致这一极致尾延迟的升高，进而带来用户体验的下降。本文首先对现有实时多媒体传输在互联网体系结构的组成成分及角色进行了分析，并提出了控制通路延迟的重要性。在网络这样一个持续波动的传输载体下，需要端上频繁地做出决策。如果端上的决策因为控制通路延迟过高而做得慢了些，都会导致极致尾延迟的升高。本文将控制通路拆分为反馈和决策两个组成部分，并分别进行优化。本文同时强调了现有数据通路架构满足极致尾延迟需求的困难，并从应用层、传输层和网络层分别对延迟波动的原因进行了定位及优化。

本文的主要研究内容和贡献可以归纳为以下几点：

1. 在控制通路上，针对由于反馈延迟波动带来的端到端延迟波动，提出了缩短反馈回环的拥塞信号早反馈解决方案 Zhuge。 Zhuge 通过将反馈回环与数据通路进行解耦，从而实现了缩短反馈回环的拥塞信号早反馈的目的。具体而言，该工作对实时多媒体传输协议，根据其反馈模式分类为带内反馈与带外反馈，并针对性地对不同类型的反馈模式分别进行了优化。基于真实路由器和大规模仿真的实验表明，第4章提出的缩短反馈回环的拥塞信号早反馈的解决方案可以有效地减少端到端延迟波动，从而提高了用户体验。该工作发表在 ACM SIGCOMM 2022 会议上，并在阿里巴巴进行了产品部署的测试，取得了良好的性能提升。

2. 在控制通路上，针对由于决策延迟及结果不稳定带来的端到端延迟波动，提出了轻量、可靠的速率控制决策框架 Metis。 Metis 是一种轻量、可靠的速率控制决策的转化与解释框架，通过将优化后的复杂的速率控制决策算法转化为简单

的速率控制决策算法，从而实现了决策的及时性与可靠性。具体而言，该工作将现有基于机器学习、整数规划等复杂的速率控制决策算法转化为基于决策树简单的速率控制决策算法。基于现有算法的实验与分析表明，第5章提出的轻量、可靠的速率控制决策的转化与解释框架可以有效地减少性能波动，从而提高了用户体验。该工作发表在 ACM SIGCOMM 2020 会议上，并在腾讯、快手等公司均进行了真实生产环境的测试部署。

3. 在数据通路上，针对由于应用层中视频编解码器延迟波动带来的端到端延迟波动，提出了自适应帧率调整的解决方案 AFR。AFR 是一种自适应帧率调整的解决方案，通过主动调整应用层中视频编解码器的帧率，从而实现了应用层中视频编解码器的延迟波动的减小。具体而言，该工作基于网络状况和应用状况的联合分析，通过排队论及随机过程的建模分析，提出了一种基于应用层的主动队列管理的解决方案。面向大规模用户的实验表明，第6章提出的自适应帧率调整的解决方案在云游戏应用中可以有效地减少端到端延迟波动。该工作发表在 USENIX NSDI 2023 会议上，并已经在腾讯规模部署两年。

4. 在数据通路上，针对由于传输层丢包及其恢复机制带来的端到端延迟波动，提出了综合多种丢包恢复机制的联合恢复方案 Hairpin。 Hairpin 是一种综合现有丢包恢复机制，特别是重传和冗余恢复这两种机制，的联合丢包恢复方案。具体而言，该工作采用马尔科夫链对丢包及重传进行联合建模，提出了一种最优的添加冗余以及决定是否重传的策略。基于真实网络数据集的实验表明，第7章提出的联合丢包恢复方案可以有效地减少端到端延迟波动，同时也降低了带宽开销的成本。该工作目前在 USENIX NSDI 的 one-shot revision 阶段。

5. 在数据通路上，针对由于网络层多应用突发竞争排队带来的端到端延迟波动，提出了一种新的路由器队列管理方案 Confucius。 Confucius 是一种新的路由器队列管理方案，通过在不依赖于端侧信息的情况下对带宽分配进行差分服务优化，实现了端到端延迟波动的减小。具体而言，该工作通过观察不同流对瓶颈队列的侵占情况，来推测出不同流的延迟敏感性，从而实现了对不同流的差分服务优化。基于真实路由器及上千个网站的测试表明，第8章提出的路由器队列管理方案，在不依赖任何端侧标签等信息的情况下，可以有效地减少端到端延迟波动。

9.2　研究展望

实时多媒体是网络系统诞生数十年来经久不衰的研究课题，但其面向的应用场景越来越复杂。从网络电话到视频会议，再到云游戏、远程手术，最后到虚拟现

实、增强现实等，应用对网络的延迟需求越来越高，场景也越来越多元化。实时多媒体应用涉及一系列系统性深的课题，有些甚至都不仅仅是网络领域的科研问题，本文只解决了一部分关键的问题，但未来还有以下一些方面可以进一步探索。

1. 与操作系统进行联合优化。随着边缘节点部署的逐渐推广以及新一代无线接入网技术（如 WiFi 6 及 5G）的规模化部署，网络的净传播延迟已经越来越低。这时，在端侧的延迟瓶颈便逐渐凸显。在操作系统等端领域当然有许多优秀的研究者也在试图降低这一延迟，但需要注意的是，网络延迟事实上是**最有弹性**的延迟组成部分：网络总可以牺牲一部分吞吐来换取低延迟。因此，如果能在预感到端侧操作系统等出现延迟瓶颈时，提前对全链路的延迟预算进行规划，可以进一步降低延迟。

2. 与不同场景进行联合优化。网络层目前的指标更多与网络服务质量相关。即使卡顿率等指标事实上是在应用层视频帧粒度进行统计的，这也绝非是用户的真实体验，而只是用户体验的估计。更进一步地，不同用户由于其生理、心理状态不同以及使用应用的不同，对同一延迟、同一画质可能都会有不同的体验。如何了解到用户的真实体验，并与此结合进行优化，尤其是在这些新兴的应用场景逐渐进入人们视线之际，同样是值得进一步深入研究的方向。

从更广阔的视角来看，本文解决的延迟问题绝非仅仅能应用于实时多媒体传输。事实上，本工作在传输层、网络层以及控制通路的设计均可以迁移到其他类似有低延迟需求的应用上去。近年来，物联网、车联网等新型网络场景也给网络研究带来了巨大的机遇。本工作的低延迟优化能否应用于其他的网络场景，是否有新的挑战需要解决，也是未来值得深入探索的方向。

参考文献

[1] Cisco vni complete forecast highlights global - consumer highlights[EB/OL]. 2022. https://www.cisco.com/c/dam/m/en_us/solutions/service-provider/vni-forecast-highlights/pdf/Global_Device_Growth_Traffic_Profiles.pdf.

[2] Livingood J. Working latency —the next qoe frontier | apnic blog[EB/OL]. 2021. https://blog.apnic.net/2021/12/02/working-latency-the-next-qoe-frontier/.

[3] Mohan N, Corneo L, Zavodovski A, et al. Pruning edge research with latency shears[C]//Proc. ACM HotNets. 2020.

[4] Jones A, Sevcik P, Wetzel R. Internet connection requirements for effective video conferencing to support work from home and elearning | netforecast[EB/OL]. 2021. https://www.netforecast.com/wp-content/uploads/NFR5137-Videoconferencing_Internet_Requirements.pdf.

[5] Meng Z, Guo Y, Sun C, et al. Achieving Consistent Low Latency for Wireless Real Time Communications with the Shortest Control Loop[C]//Proc. ACM SIGCOMM. 2022.

[6] Kämäräinen T, Siekkinen M, Ylä-Jääski A, et al. A measurement study on achieving imperceptible latency in mobile cloud gaming[C]//Proc. ACM MMSys. 2017.

[7] Ivkovic Z, Stavness I, Gutwin C, et al. Quantifying and mitigating the negative effects of local latencies on aiming in 3d shooter games[C]//Proc. ACM CHI. 2015: 135-144.

[8] ArsTechnica. Nvidia gtx 1080 review: The new performance king[EB/OL]. 2016. https://arstechnica.com/gadgets/2016/05/nvidia-gtx-1080-review/4/.

[9] Shi S, Hsu C H, Nahrstedt K, et al. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming[C]//Proc. ACM Multimedia. 2011.

[10] GFXBench. 3d graphics performance of google pixel c[EB/OL]. 2017. https://gfxbench.com/device.jsp?D=Google+Pixel+C.

[11] Wimmer R, Schmid A, Bockes F. On the latency of usb-connected input devices[C]//Proc. ACM CHI. 2019: 1-12.

[12] Slivar I, Skorin-Kapov L, Suznjevic M. Cloud gaming qoe models for deriving video encoding adaptation strategies[C]//Proc. ACM Multimedia Systems Conference (MMSys). 2016.

[13] Optimizing 5g for a new class of low-latency experiences [video][EB/OL]. 2021. https://www.qualcomm.com/news/onq/2021/07/20/optimizing-5g-new-class-low-latency-experiences.

[14] Narayanan A, Ramadan E, Carpenter J, et al. A first look at commercial 5g performance on smartphones[C]//Proc. WWW. 2020.

[15] Daldoul Y, Meddour D E, Ksentini A. Performance evaluation of ofdma and mu-mimo in 802.11 ax networks[J]. Computer Networks, 2020.

[16] Bhartia A, Chen B, Wang F, et al. Measurement-based, practical techniques to improve 802.11 ac performance[C]//Proc. ACM IMC. 2017.

[17] Ghoshal M, Dash P, Kong Z, et al. Can 5g mmwave enable multi-user ar apps?[C]//Proc. PAM. 2022.

[18] Troubleshooting your stadia experience - stadia help[EB/OL]. 2021. https://support.google.com/stadia/answer/9595943.

[19] Dasari M, Kahatapitiya K, Das S R, et al. Swift: Adaptive video streaming with layered neural codecs[C]//Proc. USENIX NSDI. 2022.

[20] Zadtootaghaj S, Schmidt S, Sabet S S, et al. Quality estimation models for gaming video streaming services using perceptual video quality dimensions[C]//Proc. ACM Multimedia Systems Conference (MMSys). 2020.

[21] Bultje R S. The world's fastest vp9 decoder: ffvp9[EB/OL]. 2014. https://blogs.gnome.org/rbultje/2014/02/22/the-worlds-fastest-vp9-decoder-ffvp9/.

[22] Huang T Y, Johari R, McKeown N, et al. A buffer-based approach to rate adaptation: Evidence from a large video streaming service[C]//Proc. ACM SIGCOMM. 2014.

[23] Mao H, Netravali R, Alizadeh M. Neural adaptive video streaming with pensieve[C]//Proc. ACM SIGCOMM. 2017.

[24] Yan F Y, Ayers H, Zhu C, et al. Learning in situ: a randomized experiment in video streaming [C]//Proc. USENIX NSDI. 2020.

[25] Sarker Z, Perkins C, Singh V, et al. Rtp control protocol (rtcp) feedback for congestion control [J]. IETF RFC 8888, 2021.

[26] Schulzrinne H, Rao A, Lanphier R, et al. Real-time streaming protocol version 2.0[Z]. 2016.

[27] Zhang L, Cui Y, Pan J, et al. Deadline-aware transmission control for real-time video streaming [C]//Proc. IEEE ICNP. 2021.

[28] x264 - wikipedia[EB/OL]. https://en.wikipedia.org/wiki/X264.

[29] x265 - wikipedia[EB/OL]. https://en.wikipedia.org/wiki/X265.

[30] Wang Z, Bovik A C, Sheikh H R, et al. Image quality assessment: from error visibility to structural similarity[J]. IEEE Transactions on Image Processing, 2004.

[31] Peak signal-to-noise ratio - wikipedia[EB/OL]. 2020. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio.

[32] Fouladi S, Wahby R S, Shacklett B, et al. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads[C]//Proc. USENIX NSDI. 2017.

[33] Fouladi S, Emmons J, Orbay E, et al. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol[C]//Proc. USENIX NSDI. 2018.

[34] Spiteri K, Urgaonkar R, Sitaraman R K. Bola: Near-optimal bitrate adaptation for online videos [C]//Proc. IEEE INFOCOM. 2016.

[35] Spiteri K, Sitaraman R, Sparacio D. From theory to practice: improving bitrate adaptation in the dash reference player[C]//Proc. ACM MMSys. 2018.

[36] Li Z, Zhu X, Gahm J, et al. Probe and adapt: Rate adaptation for http video streaming at scale [J]. IEEE J. Sel. Areas Commun., 2014: 719-733.

[37] Wang C, Rizk A, Zink M. Squad: A spectrum-based quality adaptation for dynamic adaptive streaming over http[C]//Proc. ACM MMSys. 2016: 1-12.

[38] Yin X, Jindal A, Sekar V, et al. A control-theoretic approach for dynamic adaptive video streaming over http[C]//Proc. ACM SIGCOMM. 2015.

[39] Sengupta S, Ganguly N, Chakraborty S, et al. Hotdash: Hotspot aware adaptive video streaming using deep reinforcement learning[C]//Proc. IEEE ICNP. 2018: 165-175.

[40] Yi G, Yang D, Bentaleb A, et al. The acm multimedia 2019 live video streaming grand challenge [C]//Proc. ACM Multimedia. 2019: 2622-2626.

[41] Psa: Webrtc m88 release notes[EB/OL]. 2020. https://groups.google.com/g/discuss-webrtc/c/A0FjOcTW2c0/m/UAv-veyPCAAJ.

[42] Marczak B, Scott-Railton J. Move fast and roll your own crypto: A quick look at the confidentiality of zoom meetings - the citizen lab[EB/OL]. 2020. https://citizenlab.ca/2020/04/move-fast-roll-your-own-crypto-a-quick-look-at-the-confidentiality-of-zoom-meetings/.

[43] brianhu. Google meet troubleshooting playbook - network and hardware troubleshooting [EB/OL]. 2021. https://www.googlecloudcommunity.com/gc/Workspace-Product-Articles/Google-Meet-Troubleshooting-Playbook-Network-and-Hardware/ta-p/165810.

[44] Shi H, Cui Y, Qian F, et al. Dtp: Deadline-aware transport protocol[C]//Proc. APNet. 2019.

[45] Winstein K, Sivaraman A, Balakrishnan H. Stochastic forecasts achieve high throughput and low delay over cellular networks[C]//Proc. USENIX NSDI. 2013.

[46] Carlucci G, De Cicco L, Holmer S, et al. Analysis and design of the google congestion control for web real-time communication (webrtc)[C]//Proceedings of ACM International Conference on Multimedia Systems (MMSys). 2016.

[47] Zhu X, Pan R, Ramalho M A, et al. Network-Assisted Dynamic Adaptation (NADA): A Unified Congestion Control Scheme for Real-Time Media[Z]. 2020.

[48] Johansson I, Sarker Z. Self-Clocked Rate Adaptation for Multimedia[Z]. 2017.

[49] Arun V, Balakrishnan H. Copa: Practical delay-based congestion control for the internet[C]//Proc. USENIX NSDI. 2018.

[50] Dong M, Meng T, Zarchy D, et al. Pcc vivace: Online-learning congestion control[C]//Proc. USENIX NSDI. 2018.

[51] Holmer S, Shemer M, Paniconi M. Handling packet loss in webrtc[C]//2013 IEEE International Conference on Image Processing. 2013.

[52] Fong S L, Emara S, Li B, et al. Low-latency network-adaptive error control for interactive streaming[C]//Proc. ACM Multimedia. 2019.

[53] Rudow M, Yan F Y, Kumar A, et al. Streammelt: Efficient loss recovery for videoconferencing via streaming codes[C]//Proc. USENIX NSDI. 2023.

[54] Cheng Y, Cardwell N, Dukkipati N, et al. The RACK-TLP Loss Detection Algorithm for TCP [Z]. 2021.

[55] Padhye J, Firoiu V, Towsley D F, et al. Modeling tcp reno performance: a simple model and its empirical validation[J]. IEEE/ACM transactions on Networking, 2000, 8(2): 133-145.

[56] Ha S, Rhee I, Xu L. Cubic: a new tcp-friendly high-speed tcp variant[J]. ACM SIGOPS Operating Systems Review, 2008.

[57] Cardwell N, Cheng Y, Gunn C S, et al. Bbr: Congestion-based congestion control[J]. ACM Queue, 2016.

[58] Zaki Y, Pötsch T, Chen J, et al. Adaptive congestion control for unpredictable cellular networks [C]//Proc. ACM SIGCOMM. 2015.

[59] Garg N. Evaluating copa congestion control for improved video performance[EB/OL]. 2019. https://engineering.fb.com/2019/11/17/video-engineering/copa/.

[60] Carlucci G, De Cicco L, Holmer S, et al. Congestion control for web real-time communication [J]. IEEE/ACM Transactions on Networking, 2017.

[61] Sarolahti P, Kojo M, Raatikainen K. F-rto: an enhanced recovery algorithm for tcp retransmission timeouts[J]. ACM SIGCOMM Computer Communication Review, 2003.

[62] Bolot J C, Fosse-Parisis S, Towsley D. Adaptive fec-based error control for internet telephony [C]//Proc. IEEE INFOCOM. 1999.

[63] Padhye C, Christensen K J, Moreno W. A new adaptive fec loss control algorithm for voice over ip applications[C]//Proc. IEEE INFOCOM. 2000.

[64] Issue 93006: Update to media_opt_util: - code review[EB/OL]. 2020. https://webrtc-codereview.appspot.com/93006.

[65] Chen K, Wang H, Fang S, et al. Rl-afec: Adaptive forward error correction for real-time video communication based on reinforcement learning[C]//Proc. ACM MMSys. 2022.

[66] Fong S L, Khisti A, Li B, et al. Optimal streaming codes for channels with burst and arbitrary erasures[J]. IEEE Transactions on Information Theory, 2019.

[67] Krishnan M N, Shukla D, Kumar P V. Rate-optimal streaming codes for channels with burst and random erasures[J]. IEEE Transactions on Information Theory, 2020.

[68] Nichols K, Jacobson V. Controlling queue delay[J]. Communications of the ACM, 2012.

[69] Lin D, Morris R. Dynamics of random early detection[C]//Proc. ACM SIGCOMM. 1997.

[70] Feng W c, Kandlur D, Saha D, et al. Blue: A new class of active queue management algorithms [M]. Technical Report CSE-TR-387-99, University of Michigan, 1999.

[71] Feng W c, Kapadia A, Thulasidasan S. Green: proactive queue management over a best-effort network[C]//Proc. IEEE GLOBECOM. 2002.

[72] Long C, Zhao B, Guan X, et al. The yellow active queue management algorithm[J]. Elsevier Computer Networks, 2005.

[73] Spang B, Arslan S, McKeown N. Updating the theory of buffer sizing[J]. Performance Evaluation, 2021, 151: 102232.

[74] Tang J, Liu S, Xu Y, et al. Abs: Adaptive buffer sizing via augmented programmability with machine learning[C]//Proc. IEEE INFOCOM. 2022: 2038-2047.

[75] Katabi D, Handley M, Rohrs C. Congestion control for high bandwidth-delay product networks [C]//Proc. ACM SIGCOMM. 2002.

[76] Tai C H, Zhu J, Dukkipati N. Making large scale deployment of rcp practical for real networks [C]//Proc. IEEE INFOCOM. 2008.

[77] Flores M, Wenzel A, Kuzmanovic A. Enabling router-assisted congestion control on the internet [C]//Proc. IEEE ICNP. 2016.

[78] Goyal P, Agarwal A, Netravali R, et al. Abc: A simple explicit congestion controller for wireless networks[C]//Proc. USENIX NSDI. 2020.

[79] Floyd S, Jacobson V. Random early detection gateways for congestion avoidance[J]. IEEE/ACM Transactions on networking, 1993.

[80] Feng W c, Kandlur D D, Saha D, et al. Stochastic fair blue: A queue management algorithm for enforcing fairness[C]//Proc. IEEE INFOCOM. 2001.

[81] Chatranon G, Labrador M A, Banerjee S. Black: detection and preferential dropping of high bandwidth unresponsive flows[C]//Proc. IEEE ICC. 2003.

[82] Pan R, Breslau L, Prabhakar B, et al. Approximate fairness through differential dropping[J]. ACM SIGCOMM Computer Communication Review, 2003, 33(2): 23-39.

[83] Schepper K D, Briscoe B, White G. Request for comments: number 9332 Dual-Queue Coupled Active Queue Management (AQM) for Low Latency, Low Loss, and Scalable Throughput (L4S)[M/OL]. RFC Editor, 2023. https://www.rfc-editor.org/info/rfc9332. DOI: 10.17487/RFC9332.

[84] Bai W, Chen L, Chen K, et al. Information-agnostic flow scheduling for commodity data centers [C]//Proc. USENIX NSDI. 2015.

[85] Alizadeh M, Yang S, Sharif M, et al. pfabric: Minimal near-optimal datacenter transport[C]// Proc. ACM SIGCOMM. 2013.

[86] Addanki V, Apostolaki M, Ghobadi M, et al. Abm: Active buffer management in datacenters [C]//Proc. ACM SIGCOMM. 2022.

[87] Baker F, Babiarz J, Chan K H. Configuration Guidelines for DiffServ Service Classes[Z]. 2006.

[88] Tcp analysis | cs-224-lectures[EB/OL]. https://grubdragon.github.io/CS-224-Lectures/lec/lec11.html.

[89] Appenzeller G, Keslassy I, McKeown N. Sizing router buffers[J]. ACM SIGCOMM Computer Communication Review, 2004, 34(4): 281-292.

[90] Mobile C, ZTE. Powered by sa: 5g mec-based cloud game innovation practice[Z]. 2020.

[91] Alizadeh M, Kabbani A, Atikoglu B, et al. Stability analysis of qcn: the averaging principle [C]//Proc. ACM SIGMETRICS. 2011.

[92] Dong M, Li Q, Zarchy D, et al. Pcc: Re-architecting congestion control for consistent high performance[C]//Proc. USENIX NSDI. 2015.

[93] Abbasloo S, Yen C Y, Chao H J. Classic meets modern: A pragmatic learning-based congestion control for the internet[C]//Proc. ACM SIGCOMM. 2020.

[94] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners[A]. 2020.

[95] Faber T. Acc: using active networking to enhance feedback congestion control mechanisms[J]. IEEE network, 1998.

[96] Meng Z, Guo Y, Shen Y, et al. Practically deploying heavyweight adaptive bitrate algorithms with teacher-student learning[J]. IEEE/ACM Transactions on Networking, 2021.

[97] Xu D, Zhou A, Zhang X, et al. Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption[C]//Proc. ACM SIGCOMM. 2020.

[98] Høiland-Jørgensen T, Täht D, Morton J. Piece of cake: a comprehensive queue management solution for home gateways[C]//Proc. IEEE LANMAN. 2018.

[99] Liu S, Ghalayini A, Alizadeh M, et al. Breaking the transience-equilibrium nexus: A new approach to datacenter packet transport[C]//Proc. USENIX NSDI. 2021.

[100] Start - tencent cloud gaming[EB/OL]. 2020. https://start.qq.com/.

[101] Holmer S, Flodman M, Sprang E. Rtp extensions for transport-wide congestion control[EB/OL]. 2015. https://datatracker.ietf.org/doc/html/draft-holmer-rmcat-transport-wide-cc-extensions-01.

[102] Høiland-Jørgensen T, Kazior M, Täht D, et al. Ending the anomaly: Achieving low latency and airtime fairness in wifi[C]//Proc. USENIX ATC. 2017.

[103] [systemd-devel] [announce] systemd 217[EB/OL]. 2014. https://lists.freedesktop.org/archives/systemd-devel/2014-October/024662.html.

[104] [openwrt wiki] netgear wndr3800[EB/OL]. 2011. https://openwrt.org/toh/netgear/wndr3800.

[105] Iyengar J, Swett I. Quic loss detection and congestion control[J]. IETF RFC 9002, 2021.

[106] Kjerland E, Shadbolt M, Watherston A, et al. Network requirements for windows 365 | microsoft docs[EB/OL]. 2021. https://docs.microsoft.com/en-us/windows-365/enterprise/requirements-network.

[107] Shen Y. Live video transmuxing/transcoding: Ffmpeg vs twitchtranscoder, part i | twitch blog [EB/OL]. 2017. https://blog.twitch.tv/en/2017/10/10/live-video-transmuxing-transcoding-f-ffmpeg-vs-twitch-transcoder-part-i-489c1c125f28/.

[108] Di Domenico A, Perna G, Trevisan M, et al. A network analysis on cloud gaming: Stadia, geforce now and psnow[J]. Network, 2021.

[109] Rowe C, Hanson D, Craig C, et al. Microsoft teams call flows - microsoft teams | microsoft docs[EB/OL]. 2021. https://docs.microsoft.com/en-us/microsoftteams/microsoft-teams-online-call-flows.

[110] Baugher M, McGrew D, Naslund M, et al. The secure real-time transport protocol (srtp)[J]. IETF RFC 3711, 2004.

[111] Marfia G, Palazzi C E, Pau G, et al. Tcp libra: Derivation, analysis, and comparison with other rtt-fair tcps[J]. Computer Networks, 2010.

[112] Prepare your network for meet video calls - google workspace admin help[EB/OL]. 2021. https://support.google.com/a/answer/1279090.

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

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

[115] Li Z, Aaron A, Katsavounidis I, et al. Toward a practical perceptual video quality metric | netflix techblog[EB/OL]. 2016. https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652.

[116] Webrtc samples[EB/OL]. 2021. https://webrtc.github.io/samples/.

[117] Kofler I, Prangl M, Kuschnig R, et al. An h. 264/svc-based adaptation proxy on a wifi router [C]//Proc. NOSSDAV. 2008.

[118] Yeo H, Jung Y, Kim J, et al. Neural adaptive content-aware internet video delivery[C]//Proc. USENIX OSDI. 2018.

[119] Mao H, Chen S, Dimmery D, et al. Real-world video adaptation with reinforcement learning [C]//ICML Reinforcement Learning for Real Life Workshop. 2019.

[120] Chen L, Lingys J, Chen K, et al. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization[C]//Proc. ACM SIGCOMM. 2018.

[121] Jay N, Rotman N, Godfrey B, et al. A deep reinforcement learning perspective on internet congestion control[C]//Proc. ICML. 2019.

[122] Rusek K, Suárez-Varela J, Mestres A, et al. Unveiling the potential of graph neural networks for network modeling and optimization in sdn[C]//Proc. ACM SOSR. 2019.

[123] Xiao Y, Zhang Q, Liu F, et al. Nfvdeep: Adaptive online service function chain deployment with deep reinforcement learning[C]//Proc. IEEE/ACM IWQoS. 2019.

[124] Zhang M, Bai J, Li G, et al. When nfv meets ann: Rethinking elastic scaling for ann-based nfs [C]//Proc. IEEE ICNP. 2019.

[125] LeCun Y, Bengio Y, Hinton G. Deep learning[J]. Nature, 2015, 521(7553): 436.

[126] Leshno M, Lin V Y, Pinkus A, et al. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function[J]. Neural networks, 1993, 6(6): 861-867.

[127] Sutton R S, Barto A G. Reinforcement learning (second edition): An introduction[M]. MIT press, 2018.

[128] Mao H, Schwarzkopf M, Venkatakrishnan S B, et al. Learning scheduling algorithms for data processing clusters[C]//Proc. ACM SIGCOMM. 2019.

[129] Dethise A, Canini M, Kandula S. Cracking open the black box: What observations can tell us about reinforcement learning agents[C]//Proc. ACM NetAI. 2019.

[130] Zheng Y, Liu Z, You X, et al. Demystifying deep learning in networking[C]//Proc. ACM APNet. 2018.

[131] Zeiler M D, Fergus R. Visualizing and understanding convolutional networks[C]//Proc. ECCV. 2014.

[132] Bau D, Zhou B, Khosla A, et al. Network dissection: Quantifying interpretability of deep visual representations[C]//Proc. IEEE CVPR. 2017.

[133] Toneva M, Wehbe L. Interpreting and improving natural-language processing (in machines) with natural language-processing (in the brain)[C]//Proc. NeurIPS. 2019.

[134] Ribeiro M T, Singh S, Guestrin C. Semantically equivalent adversarial rules for debugging nlp models[C]//Proc. ACL. 2018.

[135] Bastani O, Pu Y, Solar-Lezama A. Verifiable reinforcement learning via policy extraction[C]// Proc. NeurIPS. 2018.

[136] Ross S, Gordon G, Bagnell D. A reduction of imitation learning and structured prediction to no-regret online learning[C]//Proc. AISTATS. 2011.

[137] Friedman J H, Olshen R A, Stone C J, et al. Classification and regression trees[J]. Wadsworth & Brooks, 1984.

[138] Huynh L N, Lee Y, Balan R K. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications[C]//Proc. ACM MobiSys. 2017.

[139] Ecarlat P. Cnn - do we need to go deeper?[EB/OL]. 2017. https://medium.com/finc-engineering/cnn-do-we-need-to-go-deeper-afe1041e263e.

[140] Deng J, Dong W, Socher R, et al. Imagenet: A large-scale hierarchical image database[C]// Proc. IEEE CVPR. 2009.

[141] Guidotti R, Monreale A, Ruggieri S, et al. A survey of methods for explaining black box models [J]. ACM Computing Surveys (CSUR), 2018.

[142] Du M, Liu N, Hu X. Techniques for interpretable machine learning[J]. Commun. ACM, 2020: 68-77.

[143] Yin C, Zhu Y, Fei J, et al. A deep learning approach for intrusion detection using recurrent neural networks[J]. IEEE Access, 2017: 21954-21961.

[144] Guo W, Mu D, Xu J, et al. Lemna: Explaining deep learning based security applications[C]// Proc. ACM CCS. 2018.

[145] Ribeiro M T, Singh S, Guestrin C. Why should i trust you?: Explaining the predictions of any classifier[C]//Proc. ACM KDD. 2016.

[146] Blockeel H, De Raedt L. Top-down induction of first-order logical decision trees[J]. Artificial intelligence, 1998, 101(1-2): 285-297.

[147] Meng Z, Wang M, Bai J, et al. Interpreting deep learning-based networking systems[C]//Proc. ACM SIGCOMM. 2020.

[148] Verma A, Murali V, Singh R, et al. Programmatically interpretable reinforcement learning[C]// Proc. ICML. 2018.

[149] Zhu H, Xiong Z, Magill S, et al. An inductive synthesis framework for verifiable reinforcement learning[C]//Proc. ACM PLDI. 2019.

[150] Mingers J. An empirical comparison of pruning methods for decision tree induction[J]. Machine learning, 1989, 4(2): 227-243.

[151] Venables W N, Ripley B D. Tree-based methods[M]//Modern Applied Statistics with S. Springer, 2002: 251-269.

[152] Riiser H, Vigmostad P, Griwodz C, et al. Commute path bandwidth traces from 3g networks: Analysis and applications[C]//Proc. ACM MMSys. 2013.

[153] Raw data - measuring broadband america[EB/OL]. 2016. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-2016.

[154] Smilkov D, Thorat N, Assogba Y, et al. Tensorflow. js: Machine learning for the web and beyond[C]//Proc. SysML. 2019.

[155] Jiang J, Sekar V, Zhang H. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive[C]//Proc. ACM CoNEXT. 2012.

[156] Dash.js[EB/OL]. 2018. https://github.com/Dash-Industry-Forum/dash.js.

[157] Elsken T, Metzen J H, Hutter F. Neural architecture search: A survey.[J]. Journal of Machine Learning Research, 2019.

[158] Wagner K. Facebook says video is huge – 100-million-hours-per-day huge[EB/OL]. 2016. https://www.vox.com/2016/1/27/11589140/.

[159] Manapragada C, Webb G I, Salehi M. Extremely fast decision tree[C]//Proc. ACM KDD. 2018.

[160] Frankle J, Carbin M. The lottery ticket hypothesis: Finding sparse, trainable neural networks [C]//Proc. ICLR. 2019.

[161] Yu H, Edunov S, Tian Y, et al. Playing the lottery with rewards and multiple languages: lottery tickets in rl and nlp[C]//Proc. ICLR. 2020.

[162] Modi C, Patel D, Borisaniya B, et al. A survey of intrusion detection techniques in cloud[J]. Elsevier Journal of network and computer applications, 2013: 42-57.

[163] Chen J, Le H M, Carr P, et al. Learning online smooth predictors for realtime camera planning using recurrent decision trees[C]//Proc. IEEE CVPR. 2016.

[164] Ba J, Caruana R. Do deep nets really need to be deep?[C]//Proc. NIPS. 2014.

[165] Hanin B, Rolnick D. Complexity of linear regions in deep networks[C]//Proc. ICML. 2019.

[166] Narodytska N, Ignatiev A, Pereira F, et al. Learning optimal decision trees with sat.[C]//Proc. IJCAI. 2018.

[167] Wu M, Hughes M C, Parbhoo S, et al. Beyond sparsity: Tree regularization of deep models for interpretability[C]//Proc. AAAI. 2018.

[168] Novak R, Bahri Y, Abolafia D A, et al. Sensitivity and generalization in neural networks: an empirical study[C]//Proc. ICLR. 2018.

[169] Meng Z, Chen J, Guo Y, et al. Pitree: Practical implementation of abr algorithms using decision trees[C]//Proc. ACM MM. 2019.

[170] Zhang X, Wang N, Ji S, et al. Interpretable deep learning under fire[C]//Proc. USENIX Security. 2020.

[171] Heo J, Joo S, Moon T. Fooling neural network interpretations via adversarial model manipulation[C]//Proc. NeurIPS. 2019.

[172] Huang C Y, Hsu C H, Chang Y C, et al. Gaminganywhere: an open cloud gaming system[C]// Proc. ACM Multimedia Systems Conference (MMSys). 2013.

[173] Guan Y, Zheng C, Zhang X, et al. Pano: Optimizing 360 video streaming with a better understanding of quality perception[M]//Proc. ACM SIGCOMM. 2019.

[174] Zhou C, Xiao M, Liu Y. Clustile: Toward minimizing bandwidth in 360-degree video streaming [C]//Proc. IEEE INFOCOM. 2018.

[175] Petrangeli S, Swaminathan V, Hosseini M, et al. An http/2-based adaptive streaming framework for 360 virtual reality videos[C]//Proc. ACM Multimedia. 2017.

[176] Holub P, Matela J, Pulec M, et al. Ultragrid: low-latency high-quality video transmissions on commodity hardware[C]//Proc. ACM International Conference on Multimedia. 2012.

[177] Li T, Zheng K, Xu K, et al. Tack: Improving wireless transport performance by taming acknowledgments[C]//Proc. ACM SIGCOMM. 2020.

[178] Stadia - one place for all the ways we play[EB/OL]. 2020. https://stadia.google.com/.

[179] Cloud gaming (beta) with xbox game pass | xbox[EB/OL]. 2020. https://www.xbox.com/en-US/xbox-game-pass/cloud-gaming.

[180] Your games. your devices. play anywhere | nvidia geforce now[EB/OL]. 2020. https://www.nvidia.com/en-us/geforce-now/.

[181] Bhutani A, Wadhwani P. Cloud gaming market share forecast 2025 | industry size report [EB/OL]. 2020. https://www.gminsights.com/industry-analysis/cloud-gaming-market.

[182] OPG609. List of 60fps games playable on ps5.[EB/OL]. 2020. https://www.reddit.com/r/PS5/comments/kiuh2t/list_of_60fps_games_playable_on_ps5/.

[183] Han J, Smith B. Cu-seeme vr immersive desktop teleconferencing[C]//Proc. ACM Multimedia. 1997: 199-207.

[184] Nikolaevskiy I. Refactor framebuffer to store decoded frames history separately (i82be0eb3) • gerrit code review[EB/OL]. 2019. https://webrtc-review.googlesource.com/c/src/+/116686.

[185] Winstein K, Balakrishnan H. Mosh: An interactive remote shell for mobile clients[C]//Proc. USENIX ATC. 2012.

[186] Youtube vr - home - youtube vr[EB/OL]. 2021. https://vr.youtube.com/.

[187] Vr-interactive –we are interactive[EB/OL]. 2021. https://vr-interactive.at/.

[188] Facebook 360 video[EB/OL]. 2021. https://facebook360.fb.com/.

[189] Huawei video conferencing platform —huawei enterprise[EB/OL]. 2021. https://e.huawei.com/en/solutions/enterprise-collaboration/videoconferencing-platform.

[190] Meeting and phone statistics –zoom help center[EB/OL]. 2021. https://support.zoom.us/hc/en-us/articles/202920719-Meeting-and-phone-statistics.

[191] Stringer J. Pushing it to the limit – parsec at 240 frames per second with approximately 4-8 milliseconds of ... | parsec[EB/OL]. 2022. https://parsec.app/blog/parsec-game-streaming-total-latency-at-240-frames-per-second-c0818cc0daa5.

[192] Google meet and default video resolution - google meet community[EB/OL]. 2021. https://support.google.com/meet/thread/58039897/google-meet-and-default-video-resolution.

[193] Pennington A. So you say you're planning a 16k live stream... - nab amplify[EB/OL]. 2022. https://amplify.nabshow.com/articles/so-you-say-youre-planning-a-16k-live-stream/.

[194] Bulman J, Garraghan P. A cloud gaming framework for dynamic graphical rendering towards achieving distributed game engines[C]//Proc. USENIX HotCloud. 2020.

[195] Tan Z, Li Y, Li Q, et al. Supporting mobile vr in lte networks: How close are we?[J]. Proc. ACM SIGMETRICS, 2018.

[196] Carrascosa M, Bellalta B. Cloud-gaming: Analysis of google stadia traffic[A]. 2020.

[197] Mossad O, Diab K, Amer I, et al. Deepgame: Efficient video encoding for cloud gaming[C]// Proc. ACM Multimedia. 2021.

[198] Schwarz H, Marpe D, Wiegand T. Overview of the scalable video coding extension of the h. 264/avc standard[J]. IEEE Transactions on circuits and systems for video technology, 2007.

[199] Dobrian F, Sekar V, Awan A, et al. Understanding the impact of video quality on user engagement[C]//Proc. ACM SIGCOMM. 2011.

[200] Sabet S S, Schmidt S, Zadtootaghaj S, et al. A latency compensation technique based on game characteristics to mitigate the influence of delay on cloud gaming quality of experience[C]// Proceedings of the 11th ACM Multimedia Systems Conference. 2020: 15-25.

[201] Ball M, Navok J. Challenge #3: Enormous bandwidth costs and operational burdens | cloud gaming: Why it matters and the games it will create[EB/OL]. 2020. https://www.matthewball.vc/all/cloudmiles.

[202] Processor benchmarks - geekbench browser[EB/OL]. 2022. https://browser.geekbench.com/processor-benchmarks/.

[203] Gfxbench - unified graphics benchmark based on dxbenchmark (directx) and glbenchmark (opengl es)[EB/OL]. 2022. https://gfxbench.com/result.jsp.

[204] Prakash A, Amrouch H, Shafique M, et al. Improving mobile gaming performance through cooperative cpu-gpu thermal management[C]//Proc. ACM/EDAC/IEEE Design Automation Conference (DAC). 2016: 1-6.

[205] Lee Y G, Song B C. An intra-frame rate control algorithm for ultralow delay h. 264/advanced video coding (avc)[J]. IEEE Transactions on Circuits and Systems for Video Technology, 2009: 747-752.

[206] Dimitri P. Bertsekas R G G. Section 3.3: The m/m/1 queuing system[C]//Data Networks (2nd Edition). 1992.

[207] Trtx 2080 ti vs rtx 3080 ti game performance benchmarks (i7-8700k vs core i9-10900k) - gpucheck united states / usa[EB/OL]. 2021. https://www.gpucheck.com/compare/nvidia-geforce-rtx-2080-ti-vs-nvidia-geforce-rtx-3080-ti/.

[208] Zhang W, Qian F, Han B, et al. Deepvista: 16k panoramic cinema on your mobile device[C]// Proceedings of the Web Conference. 2021: 2232-2244.

[209] Lin C, Guo J, Chang H, et al. A 160kgate 4.5 kb skram h. 264 video decoder for hdtv applications [C]//Proc. IEEE ISSCC. 2006: 1596-1605.

[210] Chuang T D, Tsung P K, Lin P C, et al. A 59.5 mw scalable/multi-view video decoder chip for quad/3d full hdtv and video streaming applications[C]//Proc. IEEE ISSCC. 2010: 330-331.

[211] Zhou D, Zhou J, He X, et al. A 530 mpixels/s 4096x2160@ 60fps h. 264/avc high profile video decoder chip[J]. IEEE Journal of Solid-State Circuits, 2011, 46(4): 777-788.

[212] Zhou D, Zhou J, Zhu J, et al. A 2gpixel/s h. 264/avc hp/mvc video decoder chip for super hi-vision and 3dtv/ftv applications[C]//Proc. IEEE International Solid-State Circuits Conference. 2012: 224-226.

[213] Zhou D, Wang S, Sun H, et al. An 8k h. 265/hevc video decoder chip with a new system pipeline design[J]. IEEE Journal of Solid-State Circuits, 2017, 52(1): 113-126.

[214] Lottarini A, Ramirez A, Coburn J, et al. vbench: Benchmarking video transcoding in the cloud [C]//Proc. ASPLOS. 2018: 797-809.

[215] Bruce J, Mrak M, Weerakkody R. Testing av1 and vvc - bbc r&d[EB/OL]. 2019. https://www.bbc.co.uk/rd/blog/2019-05-av1-codec-streaming-processing-hevc-vvc.

[216] Azad S, Song W, Tjondronegoro D. Bitrate modeling of scalable videos using quantization parameter, frame rate and spatial resolution[C]//Proc. IEEE ICASSP. 2010: 2334-2337.

[217] multithreading - pin processor cpu isolation on windows - stack overflow[EB/OL]. 2022. https://stackoverflow.com/questions/15324586/pin-processor-cpu-isolation-on-windows.

[218] Li Y, Miao R, Liu H H, et al. Hpcc: High precision congestion control[M]//Proc. ACM SIGCOMM. 2019.

[219] Kingman J, Atiyah M. The single server queue in heavy traffic[J]. Oper. Manage., Critical Perspect. Bus. Manage, 2003.

[220] Jacobson V. Congestion avoidance and control[C]//Proc. ACM SIGCOMM. 1988.

[221] Pukelsheim F. The three sigma rule[J]. The American Statistician, 1994.

[222] Sargent M, Chu J, Paxson D V, et al. Computing TCP's Retransmission Timer[Z].

[223] Bridge K, Satran M. Multitasking - win32 apps | microsoft docs[EB/OL]. 2018. https://docs.microsoft.com/en-us/windows/win32/procthread/multitasking.

[224] Liu C. Hardware decoding vs software decoding in 4k h264/h265 video[EB/OL]. 2020. https://www.macxdvd.com/mac-video-converter-pro/hardware-decoding-4k-ultra-hd-video.htm.

[225] Zhang X, Chen H, Zhao Y, et al. Improving cloud gaming experience through mobile edge computing[J]. IEEE Wireless Communications, 2019.

[226] Zadtootaghaj S, Schmidt S, Möller S. Modeling gaming qoe: Towards the impact of frame rate and bit rate on cloud gaming[C]//Proc. IEEE International Conference on Quality of Multimedia Experience (QoMEX). 2018.

[227] Meng Z, Wang T, Shen Y, et al. Enabling high quality real-time communications with adaptive frame-rate[C]//To appear at USENIX NSDI. 2023.

[228] Ross E, Parente J, Jacobs M, et al. typeperf | microsoft docs[EB/OL]. 2017. https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/typeperf.

[229] S.P. K. What is hairpin net shot in badminton? - quora[EB/OL]. 2020. https://www.quora.com/What-is-hairpin-net-shot-in-badminton/answer/Kaarmukilan-S-P.

[230] van Otterlo M, Wiering M. Reinforcement learning and markov decision processes[M]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012: 3-42.

[231] Pareto front - wikipedia[EB/OL]. https://en.wikipedia.org/wiki/Pareto_front.

[232] Haq O, Raja M, Dogar F R. Measuring and improving the reliability of wide-area cloud paths [C]//Proc. WWW. 2017.

[233] Chen W, Ma L, Shen C C. Congestion-aware mac layer adaptation to improve video teleconferencing over wi-fi[C]//Proceedings of ACM Multimedia Systems Conference (MMSys). 2015.

[234] Roca V, Peltotalo J, Lacan J, et al. Reed-Solomon Forward Error Correction (FEC) Schemes [M]. RFC Editor, 2009.

[235] Roca V, Cunche M, Lacan J, et al. Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME[Z]. 2013.

[236] Zanaty M, Singh V, Begen A C, et al. RTP Payload Format for Flexible Forward Error Correction (FEC)[Z]. 2019.

[237] Jain M, Dovrolis C. End-to-end available bandwidth: Measurement methodology, dynamics, and relation with tcp throughput[C]//Proc. ACM SIGCOMM. 2002.

[238] Finite element method - wikipedia[EB/OL]. 2021. https://en.wikipedia.org/wiki/Finite_element_method.

[239] Meng Z, Kong X, Chen J, et al. Hairpin: Rethinking packet loss recovery in edge-based interactive video streaming[C/OL]//One-shot revision at USENIX NSDI. 2023. https://drive.google.com/file/d/1UyQZ7KbVv7VYhuV1brAlRxN1GCHBRkBv/view?usp=sharing.

[240] Zhang S. Soonyangzhang/webrtc-gcc-ns3: test google congestion control on ns3[EB/OL]. 2020. https://github.com/SoonyangZhang/webrtc-gcc-ns3.

[241] Schulzrinne H, Casner S L, Frederick R, et al. RTP: A Transport Protocol for Real-Time Applications[Z]. 2003.

[242] Blanton E, Paxson D V, Allman M. TCP Congestion Control[Z]. 2009.

[243] Baldantoni L, Lundqvist H, Karlsson G. Adaptive end-to-end fec for improving tcp performance over wireless links[C]//Proc. IEEE ICC. 2004.

[244] Flach T, Dukkipati N, Terzis A, et al. Reducing web latency: the virtue of gentle aggression [C]//Proc. ACM SIGCOMM. 2013.

[245] Recommendations I. G.1070 : Opinion model for video-telephony applications[EB/OL]. 2018. https://www.itu.int/rec/T-REC-G.1070.

[246] Briscoe B, Schepper K D, Bagnulo M, et al. Request for comments: number 9330 Low Latency, Low Loss, and Scalable Throughput (L4S) Internet Service: Architecture[M/OL]. RFC Editor, 2023. https://www.rfc-editor.org/info/rfc9330. DOI: 10.17487/RFC9330.

[247] Chen L, Chen K, Bai W, et al. Scheduling mix-flows in commodity datacenters with karuna [C]//Proc. ACM SIGCOMM. 2016.

[248] Lucas J M, Saccucci M S. Exponentially weighted moving average control schemes: properties and enhancements[J]. Technometrics, 1990, 32(1): 1-12.

[249] Shreedhar M, Varghese G. Efficient fair queueing using deficit round robin[C]//Proc. ACM SIGCOMM. 1995.

[250] Bajpai V, Eravuchira S J, Schönwälder J. Dissecting last-mile latency characteristics[J]. ACM SIGCOMM Computer Communication Review, 2017, 47(5): 25-34.

[251] Fontugne R, Shah A, Cho K. Persistent last-mile congestion: not so uncommon[C]//Proc. ACM IMC. 2020: 420-427.

[252] Dhamdhere A, Clark D D, Gamero-Garrido A, et al. Inferring persistent interdomain congestion [C]//Proc. ACM SIGCOMM. 2018: 1-15.

[253] Har (file format) - wikipedia[EB/OL]. https://en.wikipedia.org/wiki/HAR_(file_format).

[254] Mishra A, Sun X, Jain A, et al. The great internet tcp congestion control census[C]//Proc. ACM Sigmetrics. 2020.

[255] Fejes F, Gombos G, Laki S, et al. Who will save the internet from the congestion control revolution?[C]//Proceedings of the 2019 Workshop on Buffer Sizing. 2019: 1-6.

[256] Mittal R, Agarwal R, Ratnasamy S, et al. Universal packet scheduling[C]//Proc. USENIX NSDI. 2016.

[257] MacGregor M H, Shi W. Deficits for bursty latency-critical flows: DRR++[C]//Proc. IEEE International Conference on Networks (ICON). 287-293.

[258] Zhang J, Dong E, Meng Z, et al. Wisetrans: Adaptive transport protocol selection for mobile web service[C]//Proceedings of the Web Conference. 2021.

[259] Dukkipati N, Refice T, Cheng Y, et al. An argument for increasing tcp's initial congestion window[J]. ACM SIGCOMM Computer Communication Review, 2010: 26-33.

[260] Arun V, Alizadeh M, Balakrishnan H. Starvation in End-to-End Congestion Control[C]//Proc. ACM SIGCOMM. 2022.

[261] Arun V. Implementation of the copa congestion control algorithm using ccp[EB/OL]. 2020. https://github.com/venkatarun95/ccp_copa.

[262] Estan C, Varghese G. New directions in traffic measurement and accounting[C]//Proc. ACM SIGCOMM. 2002: 323-336.

[263] Zuo X, Cui Y, Wang X, et al. Deadline-aware multipath transmission for streaming blocks[C]// Proc. IEEE INFOCOM. 2022: 2178-2187.

[264] Meng Z, Atre N, Xu M, et al. Confucius queue management: Be fair but not too fast[C/OL]// Submission. 2023. https://drive.google.com/file/d/1EPAAkP5gTXXdNxP-YZvj-1lUKnZPb-04/view?usp=sharing.

致　谢

自 2016 年踏入 4-204，开始从事网络研究以来，不知不觉已是七年。首先我想感谢 3-229 和 4-204，我在这里疯过、累过、笑过、痛过。如今 3-229 即将改造，4-204 也不知何去何从。无论未来实验室位于何方，它们会永远在我心里。

感谢我的导师徐明伟教授。徐老师永远是我可以依靠的后盾。感谢徐老师在科研上的指导与关怀，在我有任何想法时对我的支持，在我生活面临痛苦和困难时给予我疏导关怀，在疫情时克服困难坚定支持我交流访学，在论文撰写期间对我无微不至的鼓励。每一个和您开大会、开小会的瞬间，我都会记忆犹新。

感谢一路上指导我的长辈们。以时间为序，清华大学的毕军教授严慈相济地让我在科研中很早就走上了正轨；华为的孙晨博士手把手教我如何把一个项目从 idea 转换成论文、再转换为落地的产品；纽约州立大学布法罗分校的胡宏新副教授带我在学术上和求职上打开了国际化的大门；麻省理工学院的 Mohammad Alizadeh 副教授让我见识到了什么是真正的智慧；腾讯的韩瑞先生让我看见了业界的思考；卡内基梅隆大学的 Justine Sherry 副教授则让我充分感受到了亦师亦友的温暖。

感谢我的所有合作者，包括清华大学、麻省理工学院、北京大学、中科院计算所、卡内基梅隆大学、加州大学圣塔芭芭拉分校，腾讯、阿里、华为、谷歌、百度、快手、美团等诸多合作者，在此不一一列出。没有你们我不可能完成这些工作。

感谢家人们和师兄师姐们。感谢父母一路对我的坚定不移的支持与信任，让我可以并有信心做任何我想做的事情。感谢张佳师姐在我联系导师时的耐心、我刚进组融入实验室的帮助、以及在我生活遇到困难时的关怀。感谢李江师兄和我打球的每一个早晨、和我约饭、爬山、旅行的每一个瞬间。没有你们，就不可能会有今天的我。感谢我的本科室友、研究生室友，你们也一直是我内心的支柱。

感谢孔啸、王廷风、陈婧、张艺璇、郭雅宁、沈逸昕、朱昱熹等同学，是与你们的合作教会了我要如何参与一个项目。感谢你们给了我参与的机会，让我能够从另一个视角去推进项目。可能有很多我做的不够好的地方，也感谢你们的理解。

最后要感谢这个时代。2018 年我开始从事多媒体传输研究的时候，新冠疫情还没开始，更没有人能预料到，实时多媒体传输是我们生活的一部分。有言道，“站在风口上，猪都能飞”。我的成绩的取得，很大一部分要归功于时代的机遇。感谢时代这个风口，让我的工作能够为改善人们的生活贡献一份力量。希望在不远的未来，我能够进一步为实时多媒体传输，及其衍生出来的面向未来的应用，贡献自己的力量。也希望在未来，有一个由实时多媒体传输驱动的更好的时代。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间完成的相关学术成果

个人简历

1999 年 11 月出生于辽宁省沈阳市。

2015 年 8 月考入清华大学电子工程系电子信息科学与技术专业，2019 年 7 月本科毕业并获得工学学士学位。

2019 年 8 月免试进入清华大学网络科学与网络空间研究院攻读博士至今。

2018 年 6 月至 9 月在麻省理工学院计算机科学与人工智能实验室交流学习。

2022 年 2 月至 12 月在卡内基梅隆大学计算机科学系交流学习。

在学期间完成的相关学术成果

学术论文：

[1] **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**). [CCF-A 类会议]

[2] **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**). [CCF-A 类会议]

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

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

[5] **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**). [CCF-A 类会议]

[6] **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. [CCF-A 类期刊]

[7] 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**). [CCF-A 类会议]

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

[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**). [CCF-A 类会议]

[10] 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**). [CCF-A 类会议]

[11] 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**). [CCF-B 类会议]（IWQoS 2021 最佳论文奖）

[12] 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**). [CCF-B 类会议]

[13] 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**). [CCF-B 类会议]

[14] 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. [CCF-A 类期刊]

专利：

[15] 徐明伟, **孟子立**, 王敏虎, 白家松. 一种基于深度学习的全局视野网络系统的解释方法. ZL202010532906.3 (授权日期: 2022 年 9 月 9 日).

[16] 徐明伟, 张佳, 董恩焕, **孟子立**, 杨芫. 针对移动 Web 服务的自适应传输协议选择方法和装置. ZL202011444788.7 (授权日期: 2021 年 12 月 24 日).

[17] 徐明伟, **孟子立**, 陈婧, 郭雅宁, 孙晨. 一种视频播放方法、视频播放器及计算机存储介质. ZL201910871317.5 (授权日期: 2020 年 8 月 21 日).

参与项目：

[18] 国家杰出青年科学基金。大规模网络路由理论与技术。

[19] 腾讯委托项目。基于人工智能的云游戏码率自适应方案。

[20] 阿里委托项目。面向时变、异构移动网络的实时音视频传输技术研究。

指导教师学术评语

论文研究实时多媒体传输延迟优化，具有重要的理论意义和实用价值。

论文的主要研究工作和创新性成果包括：

1. 提出了一种实时多媒体传输体系结构，分析了控制通路延迟和数据通路延迟的波动原因，给出了实时多媒体传输优化建议。

2. 针对实时多媒体传输控制通路延迟波动问题，将控制通路分解为反馈和决策两阶段，提出了控制通路和数据通路分离的反馈机制 Zhuge 和将复杂算法转化为低延迟可解释决策树的机制 Metis。实验表明，Zhuge 和 Metis 能够将控制通路不稳定带来的延迟抖动减少最多 75

3. 针对实时多媒体传输的数据通路延迟波动问题，在应用层、传输层和网络层分别提出了对应用层解码器前队列进行主动帧率管理的机制 AFR、综合重传与冗余的联合丢包恢复机制 Hairpin 和渐进式主动队列管理机制 Confucius。实验表明，AFR、Hairpin 和 Confucius 分别将实时多媒体传输延迟抖动降低 13-67

论文工作表明作者掌握了本学科坚实宽广的基础理论和系统深入的专门知识，独立从事科学研究工作的能力强。论文结构清晰，内容翔实，是一篇优秀的博士论文。

答辩委员会决议书

传输性能对互联网实时多媒体应用至关重要。论文研究实时多媒体传输延迟优化，具有重要的理论意义和实用价值。

论文的主要研究工作和创新性成果包括：

1. 提出了一种实时多媒体传输体系结构，分析了控制通路延迟和数据通路延迟的波动原因，设计了实时多媒体传输不同平面和不同协议层次的优化框架。

2. 针对实时多媒体传输控制通路延迟波动问题，将控制通路分解为反馈和决策两阶段，提出了控制通路和数据通路分离的反馈机制和将复杂算法转化为低延迟可解释决策树的机制。实验表明，所提机制能够显著减少控制通路不稳定带来的延迟抖动。

3. 针对实时多媒体传输数据通路延迟波动问题，在应用层、传输层和网络层分别提出了对应用层解码器前队列进行主动帧率管理的机制、综合重传与冗余的联合丢包恢复机制和渐进式主动队列管理机制。实验表明，所提机制可以显著降低实时多媒体传输延迟抖动，提升视频传输质量。

论文工作表明作者掌握了本学科坚实宽广的基础理论和系统深入的专门知识，独立从事科学研究工作的能力强。论文结构合理，内容翔实，达到了博士学位论文的要求，是一篇优秀的博士学位论文。

答辩过程中表述清楚，回答问题正确。答辩委员会经无记名投票一致同意通过博士学位论文答辩，并建议授予孟子立工学博士学位。