移动网络中交互直播的主播端传输优化

（申请清华大学工程硕士专业学位论文）

培 养 单 位 ： 计 算 机 科 学 与 技 术 系

工 程 领 域 ： 计 算 机 技 术

申 请 人 ： 任 青 妹

指 导 教 师 ： 崔 勇 教 授

二〇一八年五月

Transmission Optimization for Mobile Broadcasters in Personalized Live Video Streaming

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the professional degree of

Master of Engineering

by

Ren Qingmei

(Computer Technology)

Thesis Supervisor : Professor Cui Yong

May, 2018

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。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

移动设备的大量普及以及通信技术的发展让用户不再满足于单纯的观看视频直播，用户逐渐成为直播内容的贡献者。传统的流媒体直播技术，不能有效地应对无线网络的带宽抖动，也不能满足交互直播端到端时延短以及高交互的性能要求。现有的研究侧重于对直播系统架构和用户直播行为规律性的研究，优化直播传输质量的研究较为稀少。随着移动直播用户量的爆炸式增长，优化直播传输的质量至关重要。

首先，本文通过对斗鱼和Twitch等商业直播平台主播端性能的测量，发现当无线网络的带宽发生抖动时，所有平台的主播端都会观测到较长时间的丢帧现象，严重损害了观众的用户体验质量。为了找出丢帧产生的原因，本文通过分析开源直播软件的源码，发现丢帧效应产生的主要原因有两点，视频编码机制带来的帧间依赖问题，以及网络质量不佳时视频数据产生速率高于网络可用容量。现有的视频编码方案都工作在差别编码模式，一个图片组后面的帧只有依赖于前面的帧才能正常解码，因此短时间的带宽抖动会在应用层产生丢帧放大效应。另外，对于长时间的带宽波动，目前的商业直播平台无法工作在变码率模式下，恒定的视频产生速度显然无法适应复杂的移动网络。

随后，为了减少丢帧现象的发生，本文协同设计了一整套的解决方案：(1) 最优的关键帧间隔选择策略； (2) 简单且有效的智能丢帧策略 GreedyDrop；(3) GoP粒度的码率自适应策略GVBR。关键帧间隔在选择时需要综合考虑视频帧间的依赖关系以及视频压缩比，在丢帧效应和视频质量之间寻求一个平衡点。为了最大化数据有效传输量，当视频帧队列溢出时，GreedyDrop 选择性的丢弃旧的图片组；GreedyDrop 不仅保证了视频帧的解码约束以及带宽容量约束，还缓解了商业直播平台的丢帧现象。GVBR算法综合考虑了实时码率收益，码率切换损失，以及丢帧带来的质量损失，基于带宽估计值和视频队列数据量之间的差值来选择合适的码率，通过调节系数为队列缓存小的主播端定制化策略。

最后，为了验证解决方案的有效性，本文选取了三个最新的码率自适应算法作为对比算法，在LTE和Wifi网络环境下分别进行了仿真。实验结果表明，本文提出的解决方案在维持相同码率的基础上，大幅度缓解了主播端的丢帧现象，减少了用户上传失败的发生。

关键词：交互直播；主播端；用户体验质量；移动网络；自适应码率

Abstract

With the popularization of mobile devices and the rapid development of communication technologies, instead of simply watching live video, users more and more participate in contributing the live streaming. Traditional streaming technology cannot effectively cope with the bandwidth jitter of the wireless network, nor can it meet the requirements of low-latency and high interactive live streaming. Recent researches focus on the system architecture of live streaming and the pattern of users' behavior, studies on optimizing the QoE of live video transmission is rare to find. With a huge number of users, optimizing the QoE of live video is imminent.

Through measurements on multiple popular live video streaming platforms, e.g., Twitch, Douyu and so on, we find that they all suffer from broadcaster-side video quality degradation caused by unnecessarily persistent video interruptions in the presence of transient bandwidth fluctuations. Analyzing the source code of the open source live broadcast software, there are two main reasons for the frame dropping issue, the inter-frame dependency caused by video encoding, and that the video data generation rate is higher than the network capacity when the network is in bad condition. Existing encoding technologies work in delta encoding mode, where the decodability of later frames depend on the former frames in the groups of pictures. Thus a transient bandwidth drop would introduce a long-term frame drops in application level. Besides, for the long-term bandwidth fluctuation, the-state-of-art live streaming platforms cannot change their bitrate. Constant video data generation rate definitely cannot adapt to variable wireless network.

This paper takes a holistic stance, and presents a suite of solutions that optimizes the broadcaster-side QoE through (1) a keyframe placement strategy, (2) a simple-yet-efficient frame dropping strategy GreedyDrop, and (3) finally, a RTMP-based bitrate adaptation strategy GVBR. An appropriate keyframe interval can dynamically trade crossframe compression for lowered inter-frame interdependency. While GreedyDrop is designed to prevent excessive frame drops observed in many popular streaming platforms under the constraints of frame decodability and the bandwidth capacity. GVBR is customized for video broadcasters who have extremely shallow buffer (below one second). GVBR take the bitrate, rate switch penalty and frame dropping into consideration. We compare our

solution with several state-of-the-art video adaptation algorithms in a variety of network conditions, and find that our solution can substantially reduce the frame drops, while achieving comparable bitrate.

Key words: Personalized Live Video Streaming; Broadcaster; QoE; Mobile Network; Adaptive Bitrate

目 录

主要符号对照表

RTMP	实时消息传输协议 (Real Time Message Protocol)
MPC	模型预测控制理论 (Model Predictive Control)
GoP	图片组 (Group of Pictures)
SSIM	结构相似性 (Structural Similarity)
HLS	HTTP Live Streaming
DASH	Dynamic Adaptive Streaming over HTTP
GDR	Gradual Decoding Refresh
T	总决策时长
$T1$	帧在队列中的最长留存时间
C_j	第j个时刻的网络带宽
N	两个关键帧之间的总帧数
S	帧的数据量大小
R_j	第j帧的的码率
x_{ij}	第j个时刻，第i帧是否在视频队列中
y_{ij}	第j个时刻，第i帧是否被发送出去
z_{ij}	第j个时刻，第i帧是否被丢弃
M_j	第j个时刻可以发送的最大帧编号
$timespan$	丢帧算法中队列时间长度
$dropPFrame$	丢帧算法中P帧的丢帧优先级
$dropBFrame$	丢帧算法中B帧的丢帧优先级
$bandwidth$	丢帧算法中每个时隙的带宽
D_j	第j个时刻的丢帧数
N_j	第j个时刻的GoP组数
S_j	第j个时刻发送出去的GoP组数
$Rest_j$	第j个时刻队列的剩余时长
T_k^j	第j个时刻第k个GoP的剩余时长
R_k^j	第j个时刻第k个GoP的码率
η	码率选择算法的平衡参数
$Rest$	码率选择算法中的队列剩余数据量
$Send$	码率选择算法中发送的数据量
$Drop$	码率选择算法中的丢帧数

第1章　绪论

1.1　课题研究背景

根据中国互联网络信息中心（CNNIC）统计，截至2017年12月，网络直播用户规模达到4.22亿；其中，游戏直播用户规模达到2.24亿，较2016年底增加7756万，占网民总体的29%，真人秀直播用户规模达到2.2亿，较去年底增加7522万，占网民总体的28.5%[1]。而据思科公司的数据预测，2016年到2021年的5年间网络直播会增长15倍，到2021年底网络直播会占总视频流量的13%[2]。目前工业界出现了许多大型的直播平台，比如，国内有斗鱼[3]，熊猫tv[4]，国外以Twitch为首的游戏直播平台，具有社交性质的Facebook Live[5]、Periscope[6]等，如图 1.1所示。而移动网民数量的快速增长，推动直播行业的发展转移到移动端，从2016年下半年开始，移动直播用户赶超PC端用户。大量的统计数据表明，网络直播会成为互联网中越来越重要的存在，而移动直播作为其中占比较大的部分，研究如何优化移动直播的用户体验质量有着重要意义。

现在火爆的交互直播是由最开始的传统直播演变而来。传统的直播一般只针对“重大事件”，比如ESPN的篮球比赛，奥运会直播等等。传统直播一般预先知道直播的具体时间，运营商会提前为直播源分配好优质充足的链路资源，供直播源上传视频至直播源服务器；源服务器接收到直播视频流后，可能会对接收到的视频流进行一定的转码操作，将RTMP流转码为HTTP流，也可能会将原先单一码率的视频分别编码为多种码率，储存在源服务器上；之后源服务器将编码完的视频，选择合适的码率通过CDN网络，IP组播和P2P[7]等分发技术分发到每个CDN边缘节点；观看直播的用户从CDN边缘节点处获取视频。传统直播一般全程使用HTTP协议，为了应对可能出现的网络状况变化，保证用户观看的连续性，传统直播一般会把发送和接收缓存都设置得比较大，导致端到端的时延也较大，为几十秒左右。但随着移动设备的普及，用户已经不再满足于单纯的观看直播，用户开始逐渐成为直播内容的贡献者，传统直播技术此时不再能够满足用户的需求。同时由于LTE等通信技术的发展，低延迟的个人交互直播应运而生，繁荣发展。

RTMP（Real Time Message Protocol），即实时消息传送协议，用于传递音频信息以及视频信息。RTMP协议因其较低的端到端时延在交互直播中得到了广泛应用。RTMP是一种面向连接的实时流式传输协议，在传输开始前RTMP层首先要握手先建立连接，面向连接的传输协议保证了传输的安全性。消息是RTMP传

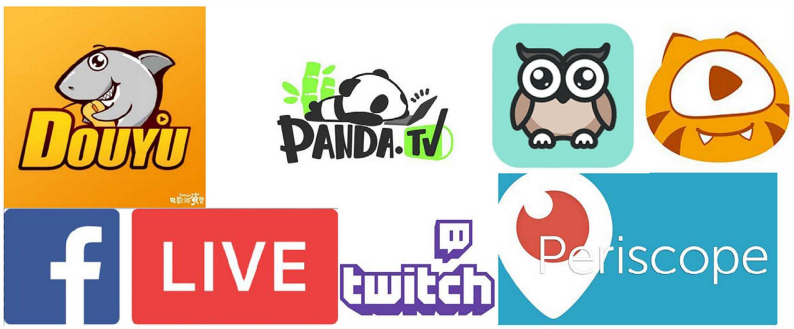


图 1.1　流行的直播商业平台

输协议的基本数据单元，消息包含音频信息和视频信息，一般将一个视频帧封装成一个视频消息。而HTTP协议的基本传输数据单元是一个视频块，视频块时长一般为4-10s，只有当完整接收到一个视频块之后，用户端才能正常播放。以视频帧作为RTMP协议的基本传输单元，RTMP协议对应的端到端延时较小。RTMP协议低延时的特性使得它被应用在许多商业直播平台上面，包括Facebook Live, Periscope,熊猫TV，斗鱼等国内外大型直播平台。

尽管学术界目前有很多研究都关注于提升直播平台的用户体验质量，但很少有人研究在直播应用中，如何通过优化主播端的上传质量，从而去优化用户的观看质量。实际上，对于直播应用来说，主播端的视频传输质量非常重要。如图3.1所示，主播端在直播架构中占据着源头的地位，所以主播端的任何延迟或者上传失败都会随着分发网络传递到用户端，影响着全部用户的观看体验质量。除此之外，主播端为用户观看到的视频传输质量设置了一个最高的限制，下游用户的质量不可能超过主播端的质量。因此，主播端需要在可能的情况下上传尽量高清的视频。

1.2　主要研究目标和贡献

个人直播应用虽然由传统直播演变而来，但也不同于传统的直播应用。传统的直播应用的源服务器处上传带宽相对充足稳定，端到端的传输时延一般会在几十秒的量级。但在个人交互直播中，大多数的主播端可能会使用无线网络进行直播，同时由于移动直播中主播端不固定的特性，网络抖动会比较剧烈，网络带宽不稳定。另外，交互直播要求端到端的时延必须在几秒以下，才能满足主播和观众互动的要求。比如，有观众给主播提问或者送礼物，主播需要及时的回应，否则可能会打击观众的积极性。交互直播的这些特性为主播端传输优化带来了新的挑战，可以总结为一句话：在抖动的带宽环境下尽量提供较高的传输质量和用户体验，包括视频质量和端到端的延时等要求。

对于个人交互直播来说，无线网络环境可能会带来新的质量问题。为了验证是否现有的直播平台能否有效的应对无线网络环境，我们首先进行了一定的测量实验。我们选择了两个流行的直播应用，斗鱼和Twitch，通过在主播端控制带宽变化，监控主播的出口吞吐量，发现这些直播平台都普遍存在着两个质量问题：

1) 短暂的网络带宽抖动在应用层会产生放大效应，从而导致长时间的视频质量下降。测量过程中，我们发现如果主播端发生了1s的网络抖动，那么应用层观测到的结果是主播端会有数秒的上传暂停的情况出现。因为交互直播要求端到端的时延尽量小，所以系统各个部分的缓存都很小，暂停数秒上传反映到观众端，则是几秒的黑屏或者暂停。
2) 现有的直播平台都无法有效应对带宽长时间降低的状况。由于设备移动的原因，可能导致小区间切换，WiFi和蜂窝网切换等问题，主播端经常会出现长时间的带宽波动。但经过我们的测量发现，目前的商业实现都不能有效的解决这个问题。目前的商业平台在带宽降低的状况下，依然维持着初始的码率，视频产生速率远远高于网络容量，从而引起严重的视频质量下降。

通过对开源直播应用源码的分析发现，应用层的放大效应是由于RTMP协议的丢包行为导致。当视频帧的队列溢出时，主播端会主动丢包，丢包导致剩下的一些视频帧无法解码，从而用户端会观测到长时间的视频卡顿。而且，一些简单直接的方案，例如，增大视频帧队列的长度，或者换用其他的丢包方式，都会带来一定的质量损失，或者增大端到端的时延，或者降低视频的质量。比如，增大主播端的视频队列长度，可以很好的解决短暂的带宽下降，但端到端的最高时延也会相应的增大。

另外，对于长时间的网络带宽降低，一个可能的解决方案是码率自适应带宽。但现有的方法都集中在研究点播场景下的自适应码率，这些在点播领域效果比较好的自适应码率方案，在主播端表现不好，因为点播情况下视频的队列长度一般维持在10s左右。而主播端的视频队列通常只有1s左右。由于更小的视频队列长度，主播端准确的码率自适应更加具有挑战性。

本文中，我们提出了一整套的解决方案，简称为GVBR（Greedy Variable Bitrate Solution），大大提高了交互直播中主播端的视频质量。我们的主要出发点是主播端的质量问题可以通过跨层协同设计来解决。GVBR主要包括三层，RTMP层，GoP级别，以及帧级别，三个级别的优化目标都是保证视频的质量和及时性。 RTMP层的配置调整主要是去选择最优的关键帧间隔，因为过小的关键帧间隔会导致视频压缩比过高，视频质量损失，而过大的关键帧间隔会导致放大效应带来的丢帧过多；GOP层面根据预测的带宽和视频帧队列数据量去自适应选

择每个GOP的码率；帧级别的解决方案主要是一个智能丢帧策略，尽可能多的减少放大效应。虽然我们的整套解决方案修改了视频流协议的两层，但所有的改动都不用修改内部逻辑，或者是更改可调的参数（例如，关键帧间隔），或者是改变软件中的控制逻辑（丢帧的逻辑和码率自适应策略）。

通过仿真，我们说明好的RTMP层协议可以大大的提高视频质量。通过改变关键帧间隔，我们测量不同关键帧间隔对应的视频质量和丢帧数，给出关键帧的最优选择范围；智能丢帧策略与离线最优算法和默认丢帧策略做对比，相比于默认策略减少了15%的丢帧，和最优策略有5%的差距。本文选择了动态码率在点播领域最新的两个算法作为对比算法，与我们提出的算法GVBR作对比。通过在不同网络环境下的大规模仿真，我们发现GVBR同目前现先进的算法比，可以减少至少50%的丢帧；而相比原始默认的算法，在保证同样视频码率的情况下，视频卡顿的概率减少90%。

总而言之，本文有两个主要的贡献点：

1) 我们是第一个把目光着眼于提升主播端视频质量的。通过对于流行视频直播平台的测量，我们发现主播端普遍存在两个质量问题，网络层短暂的带宽波动会在应用层导致长时间的视频卡顿，并且现在的直播应用都无法有效解决长时间的带宽波动。
2) 我们提出了一整套解决方案，去协同解决观测到的视频质量问题，包括帧编码参数和智能丢帧策略，以及GOP级别的比特率自适应策略。

1.3　文章组织架构

本文的内容共有六章，按照如下方式展开：

第一章是绪论部分，主要介绍交互直播的发展过程，简要概率了我们的研究目标。

第二章介绍了直播相关的一些背景知识，主要包括个人直播的架构、特点、以及性能要求。

第三章从我们搭建的平台出发，给出测量中发现的问题，目前的直播平台并不能很好的满足性能要求；并给出在商业平台中的测量，实验发现商业平台也不能很好的解决上述问题。

第四章，剖析直播软件的源码，发现上述问题出现的原因，主要是由于视频编码输出的码率不能实时的按照带宽变化，最后我们给出了一些优化设计的原则。

第五章根据设计准则，我们设计了一整套的解决方案，涵盖帧级到GoP级。

第六章与之前的算法进行比较，进行充分的实验仿真。

第七章总结现有的工作，并展望未来的研究方向。

第2章　研究现状概述

随着移动通信技术和个人设备的迅速发展，移动直播的应用呈现井喷式发展。据思科公司的数据预测，未来3年网络直播会占据总视频流量的13%。移动直播用户数量大幅度增长的同时，用户对于直播服务质量的要求也随之提高。

传统直播能够将视频通过网络分发给大量的用户，为广大用户提供视频服务。但传统直播的不足之处在于要求为主播端预留专线带宽，且端到端的时延较大。在大量用户渴望成为直播内容贡献者的今天，要求每个主播都拥有专线带宽并不可行；而且传统直播的高时延无法满足观众对于直播互动性的要求。

2.1　直播传输优化研究现状

本小节，我们通过传统直播演进到交互直播的过程，从系统架构，研究内容等方面介绍直播传输优化的研究现状。

2.1.1　直播发展现状

传统直播的系统架构和交互直播基本相同，传统直播也由三部分组成，视频源端到源服务器，源服务器到CDN边缘服务器，CDN边缘服务器到观众。但传统直播的视频源端到源服务器的链路一般为预先分配的专有链路，所以传统直播研究的重点集中在源服务器到观众端的部分。从源服务器到观众端主要有两种分发架构，CDN网络[8]和P2P网络[9][10][11]。CDN网络分发和P2P网络分发各有优缺点，CDN网络分发时会根据网络负载去选择负载轻的服务器，根据用户的带宽去选择合适的码率。当网络中服务器负载不高时，CDN网络分发能给终端用户提供较高的体验质量。但为了保证用户的体验质量，CDN网络需要保证网络各服务节点负载不高，因此需要配置相当数量的服务器，运营成本较高。相比较而言，P2P网络具有高度可扩展性，对服务器的负载要求较少，然而P2P这种去中心化的合作方式大规模应用的话会有一些缺点，比如，用户接收到的视频质量低，节点之间负载不均衡，无法进行网络监管等问题。

Yin等学者[12]提出了P2P网络和CDN网络混合分发的LiveSky架构，并在实际中部署，LiveSky的系统图如图 2.1所示。针对P2P和CDN的联合架构，Yin等人设计了一个分段阈值式的资源动态分配机制，在用户规模增大的情况下依然能提供良好的视频观看体验。同时他们利用CDN的边缘服务器和重定向机制去解

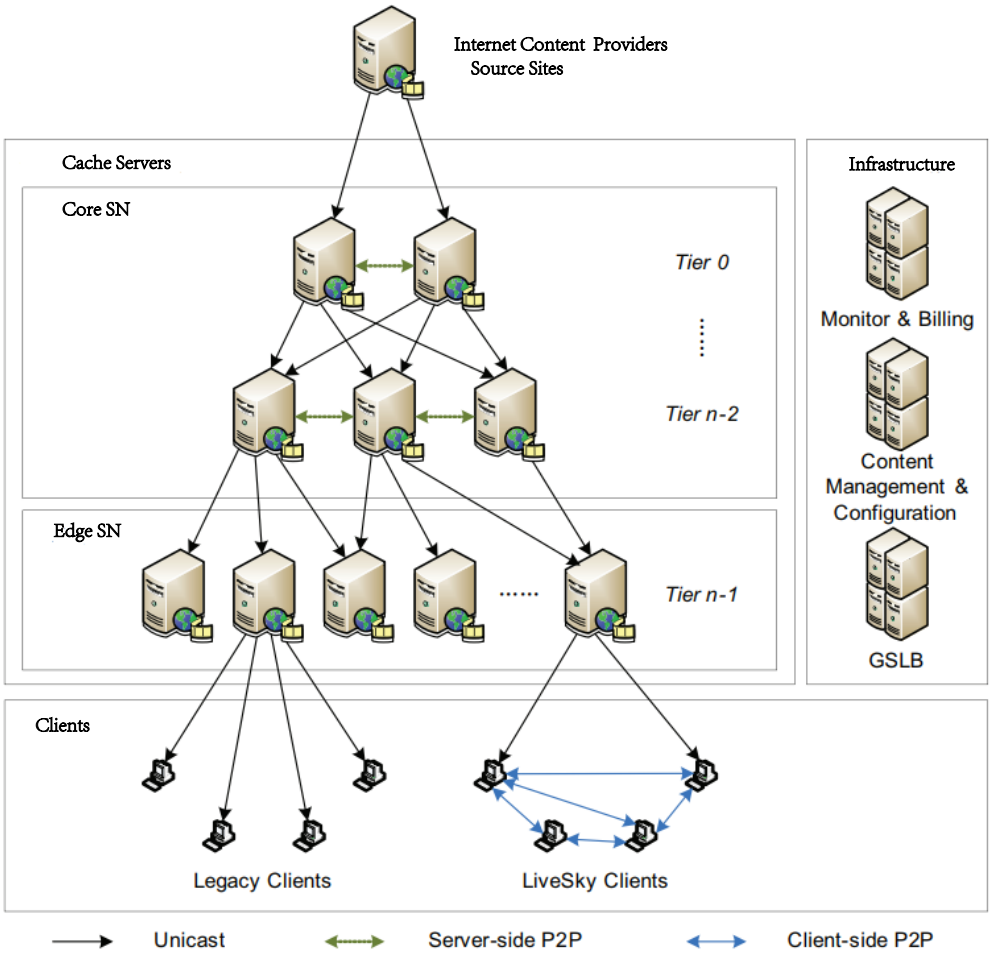


图 2.1　LiveSky系统架构图[12]

决P2P系统的不足，使得采用NAT地址转换的用户也可以完成上传，保证对现有网络而言的公平性。

学术界目前关于个人交互直播的研究多数还只是着眼于分析直播用户的行为，以及对交互直播系统框架的解剖。Zhang等学者[13]通过爬取Twitch网站[14]提供的数据，抓取本地主播端的流量勾勒出了交互直播的内部架构；另外，通过对Twitch网站数据两个月的抓取，Zhang等发现用户的观看行为基本由重大事件和主播决定。另外，观众端感受到的时延会严重影响观看的交互体验。Twitch将直播视频分发和信息发送分开来实现，因此直播消息的时延很短，视频时延很长，两者之间差别很大。另外，Zhang等学者改变主播端的带宽容量发现，主播端的带宽容量会明显影响直播视频播放的流畅性。

Meerkat和Periscope是另外两个国外主流的移动直播应用，主要部署在移动端。Matti等作者重点研究了Periscope应用的用户体验质量[15]，主要是播放流畅性，延迟和能量消耗等相关因素。Periscope的接口信息是通过HTTPS加密的方式传播的，Matti等人通过建立透明代理的方式去截取Periscope应用接口的信息。并

且Periscope的接口并不直接给出平台的总用户数信息，只提供邻近地区的用户数，Matti等人通过深度挖掘每个地区用户量的方法去组合获取实时的总用户量。另外，Periscope主播用户的粉丝数呈现长尾分布，主播用户的粉丝数会影响主播用户的积极性，粉丝数少的主播直播时长会相应的变短。

直播应用一般会给每个主播分配一个独一无二的ID，Wang等作者利用这个特性[16]，以高频率的轮询算法随机观看直播，去计算Meerkat[17]和Periscope的用户数和主播数。另外，Wang等人发现Periscope直播平台为了增加系统的可拓展性，在用户数小于100时，直接使用RTMP协议进行分发，当用户数超过100时，多出来的用户通过HTTP协议进行直播视频分发。Wang等人对两种协议情况下主播端到用户端的每一段引入的时延进行了分析，发现使用HTTP协议分发视频，相比于使用RTMP协议直接获取视频，引入了切块时延和用户请求延时。另外，使用HTTP协议进行视频分发时，用户端的缓存策略一般比较保守，缓存时间较长，从而导致较大的缓存时延。

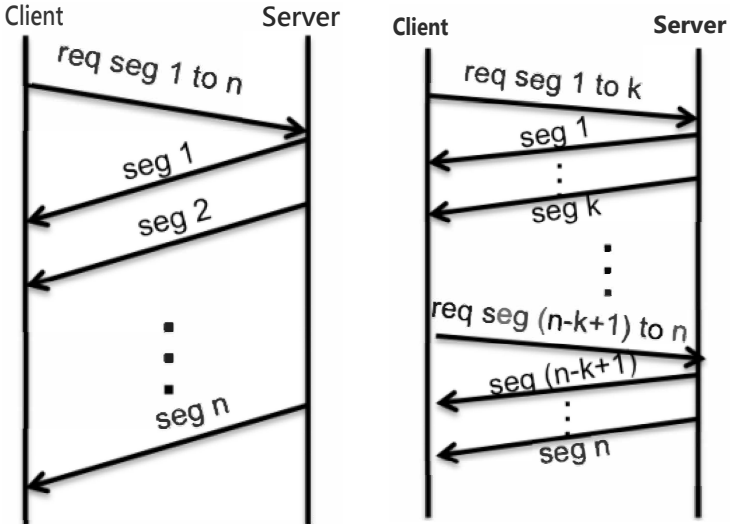
关于优化直播传输质量的研究很少，但通过优化主播端的传输质量来优化用户体验质量的研究更加稀少。

2.1.2　直播协议研究现状

在众多视频传输协议中，HTTP协议凭借广泛部署的CDN网络以及易穿透防火墙，服务器代码开源等特性，逐渐成为视频点播领域的主流协议[18]。现在的HTTP协议在视频方面主要有APPLE的HLS（HTTP Live Streaming）协议[19]和DASH（Dynamic Adaptive Streaming over HTTP）协议[20][21]，两者的原理基本相同。HTTP协议都会对视频进行切块操作，将一个完整视频切为许多4-10s的视频块，视频块是传输和处理的基本单元。HTTP协议的延时较高，主要是因为只有当一个视频块完整的产生后，CDN端才开始分发，用户端才能开始播放，所以不可避免的至少有一个视频块的延时，称为切片时延。由于切片时延的存在，且切片时延一般为数秒的量级，HTTP协议的延时一般都较大，因此HTTP协议在直播领域的应用较少。

针对HTTP协议的切块时延，一个比较直接的解决方案是减少视频块的时长，减到直播情景可以接受的时间长度，减小由于视频切块带来的时延。但是若直接减少视频块的时长，会导致HTTP请求数的倍速增长，使得HTTP服务器的负载过大。客户端发起的HTTP请求数和视频块的时长成反比，比如，对于一个60秒的直播视频，如果每个视频块是4秒，总的请求响应数目为15；但当我们把视频块的时长缩短为1秒时，总的请求响应数目会增长为60。因为每个HTTP请求和响应都

会有头部报文，倍数增长的请求响应给服务器和网络设备带来了额外的处理开销，这对HTTP视频流的可扩展性造成了极大的影响。



(a) 一次推送全部内容　　(b) 一次推送多个内容

图 2.2　HTTP协议2.0版本的新特性[22]

学术界有一些研究尝试将HTTP技术应用到直播中来，研究如何解决HTTP协议的切块延时过长问题。HTTP协议2.0版本引入的服务器推流特性允许服务器直接将资源推流到客户端，不需要客户端去明示的请求资源，这个特性极有可能打破视频块大小和请求数之间的反比关系。Wei等学者便尝试利用HTTP协议2.0的推流特性去减少切块时延[22]，因为HTTP协议2.0并不是专门的视频传输协议，因此Wei等人提出了一个定制的视频推流策略。利用HTTP服务器推流减少时延的主要思路是尽可能减少视频块的时长直到满足时延要求。推流策略需要解决的问题有两个方面：推哪些内容和什么时候推。文章给出了两种策略，一次请求推送全部内容和一次请求推送多个内容，如图 2.2。文章通过大量的实验说明，使用服务端推流的特性，直播的时延可以大大的降低。

HTTP协议的分块传输编码机制也被用来减少HTTP协议中端到端的时延[23]。Swaminathan等学者在文章中提供了两种减少端到端时延的方案，分别是服务器等待策略和分块编码策略。服务器等待的策略和原来的视频块策略相比，只做了些微的修改。最根本的改变是客户端请求正在生成的新视频块，而不是已经生成的视频块。为了达到这一目的，客户端请求描述文件中最新视频块的后面一个视频块。因为请求的视频块尚未生成，所以服务器在收到请求后一直处于等待状态，直到视频块生成为止。服务器等待的策略减少了将近一个视频块的时延，但这个优化的程度还远远不够。为了进一步优化端到端时延，Swaminathan等人进一步提

出了分块编码策略。和服务器等待策略相同，分块编码策略也总是请求后面一个视频块。分块编码策略中，服务器端将视频块切分为均匀的多个视频片段，收到客户端的请求后，将编码完成的视频片段先发送出去，之后陆续将剩余的视频片段发送出去。不同于最简单的HTTP协议，HTTP分块编码允许在服务器端维护一个HTTP长连接，客户端只需要一次请求，就可以接收数次响应消息。利用分块编码，同时将每个视频块的时长增大，分块编码达到了减少时延的效果，将时延降低为1 2个视频片段的时长。

Bouzakaria等人利用GDR（Gradual Decoding Refresh）[24]技术去编码，将直播视频编码成ISO基本媒体文件格式，并在传输层使用分块编码传输策略去减少HTTP协议的时延[25]。GDR技术是相对于用完整的一个视频帧来刷新的技术而言的。传统的完整帧刷新编码技术有一定的缺点，关键帧相对于其余的帧体积过大，容易对网络造成冲击，导致网络抖动。而GDR技术将关键帧分散到几个非关键帧中，缓解了对网络带来的冲击。Bouzakaria等人采用和Swanubathan类似的思路，改造了DASH传输协议的MPD文件，在原有MPD文件中添加了一个字段：有效时间偏差。有效时间偏差代表了新的视频块可以请求的时间，即使新的视频块可能只产生了一部分片段。有效时间偏差字段使得客户端提前得知有新的视频块产生，并提前发起请求，有效减少了HTTP协议的延迟。

上述研究都一定程度上解决了HTTP协议的低时延问题，但目前商业化平台应用仍然只在视频分发时使用HTTP协议，因此我们研究的重点依然集中在主播端使用较多的RTMP协议。

2.2　码率自适应研究现状

关于码率自适应研究的文章很多，其中大部分都集中在点播领域[26] [27] [28]。点播领域的码率自适应算法可以分为三类：基于带宽的，基于缓存的，以及两者的结合。

FESTIVE[29]就是典型的基于带宽的码率自适应算法。FESTIVE通过测量现有的商业级播放器，包括Netflix，Adobe OSMF等，发现当多个动态码率播放器共享同一链路时，会出现带宽分配不均匀的问题。另外，高码率和码率的稳定性在动态码率情况下也是应该考虑的重点。针对这三个性能指标，FESTIVE提出了一整套的解决方案。FESTIVE采用随机调度视频块的方式去减缓其他播放器下载视频占用带宽带来的带宽探测误差，从而尽量避免带宽分配的不均匀；用状态转移的方式去选择码率，拟合码率和预估带宽之间的偏差；码率延迟更新策略，以达到高码率和码率稳定性之间的平衡。

大多数码率自适应算法都是基于带宽去选择码率[30][31]，但是对于无线网络环境，尤其是在移动设备的情况下，带宽变化剧烈，很难去精确的预测带宽。Huang等人[32]首次创新性的提出了基于缓存的数据量去选择码率的算法。视频播放一般可以划分为两个阶段：启动阶段和稳定阶段。视频开始播放时，缓存的数据先从零开始积累到一个阈值，之后边下边播，缓存进入稳定阶段。基于缓存选择码率的算法基于一个想法：当缓存数据量较大时，用户端可以选择较高点的码率，提高码率的收益；如果缓存数据量较小，那用户端应该保守一些，选择较低点的码率，以保证不卡顿流畅播放。在开始阶段，缓存相关信息量较少，此时利用带宽预测的信息选择码率是很有必要的；但一旦进入稳定阶段，缓存数据量相关的信息充足时，只根据缓存去选择码率也可以性能很好。

Yin等学者[33]利用模型预测控制理论（MPC）来优化码率选择。MPC尝试预测未来一段时间的网络环境，模拟未来一段时间的系统状态，遍历求得一段时间的最优解决方案，然后只采用下一时隙的解决方案，循环往复。MPC将之前基于码率和基于缓存的方案结合起来，在优化时将码率和缓存同时纳入了优化空间，MPC的优化空间如图所示。另外，求一段时间内的最优方案有更大可能会优于某一时刻的最优。码率遍历选择的算法由于其高时间复杂度在实际运行中会面临问题，Yin等人提出了基于查询缓存结果的FastMPC算法，相对现有的所有算法提高了10%-15%的平均用户体验质量。

点播和直播领域的码率自适应主要有两点不同。第一个不同点是时间粒度，点播动态码率的粒度为一个视频块，而一个视频块持续4-10s秒时间；直播码率变化的粒度为一个GoP或者更小，一个GoP通常为2秒左右。第二个不同点是缓存队列的大小，点播时通常会缓存数十秒以防止播放的卡顿；而直播时缓存最少不超过1秒，否则会增加端到端的时延。

学术界也有一些关于直播如何使用码率自适应技术的研究[34]。Pires等人[35]通过对Twitch平台用户规模以及用户行为的研究，提出可以应用码率自适应技术分发直播流。直播的流程如图 3.1所示，如果想要应用码率自适应技术进行视频分发，需要在源服务器处进行转码，将主播端上传的单一码率视频转码为多个码率，之后用户选择自己适合的码率。Twitch平台上并发的主播数量很多，对所有主播直播的视频进行转码会需要消耗大量的计算资源。文章考虑主播的观众数，转码带来总的用户体验质量的提升以及资源消耗等因素的折中，选择部分主播进行转码操作。

Bouzakaria等学者[25]尝试用动态码率自适应技术去实现超低时延的直播。动态码率自适应技术的时延主要有几个部分构成：切块时延，下载视频块的时间，

以及用户端的缓存时长。如果将视频块切成更小的数据块在网络中传输，切块时延也会减小为小视频块的时长。之前的CDN和播放器都是在接收到一个完整的视频块之后，才开始转发以及播放。为了进一步减少时延，Bouzakaria等学者修改了视频信息描述文件，并且使用HTTP协议的分块传输编码，一旦接收到一个视频块的一段，就可以开始分发和播放，将原有的一个视频块的时延减少为一段的时延。

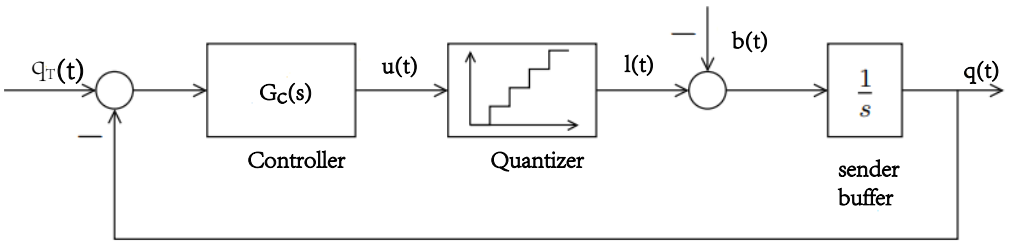


图 2.3　反馈控制理论[36]

De等人[36]用反馈控制理论去选择最优码率。不同于之前的启发式算法，反馈控制理论可以得到一个可预测的系统状况，更方便下一时隙的码率选择。控制器工作流程如下图 2.3：发送端的缓存大小作为输入，q_T是发送端缓存的阈值。控制器的激活部分，以发送端现在的缓存大小和阈值q_T的差作为输入，输出控制信号。输出的控制信号经过量化，选择最大的小于量化后的值的码率，即为最后所选择的码率。

上述关于在直播中使用码率自适应技术的研究多数关注于如何大规模部署码率自适应技术，以及超低时延的码率自适应技术，De等人的研究虽然关注于如何选择码率最优，但是研究的是点对点的直播，且端到端的延时较大。总体来说关于如何优化直播应用主播端的传输质量的研究很少，所以我们的研究也具有一定的挑战性。

2.3　交互直播的性能要求

点播情景下对于系统的性能要求一般是三点：高码率，敏捷平滑的码率切换以及低卡顿率。但主播端不同于点播客户端的根本一点在于，点播的客户端是被动接收请求的视频，而主播端负责将摄像头编码产生的视频推送出去。网络状况不好时，点播的客户端接收视频会受影响，导致视频播放出现卡顿；但对于主播端来说，网络状况不好，视频的上传会受到影响，从而导致视频帧从产生到到达用户端的时延过长，用户观看直播的过程中出现黑屏。经过上述分析，移动网络带宽不稳定的状况下，观众和主播间的高交互性要求对直播的主播端提出了如下

的性能要求。

- 尽可能高的视频质量。视频质量的优劣，和视频的码率，每秒的帧率，以及分辨率都息息相关。在帧率和分辨率固定的情况下，主播端应上传尽可能高的码率到源服务器。源头的码率高，下游的用户才有可能接收到清晰的视频，得到满意的用户体验质量；
- 敏捷平滑的码率自适应。交互直播一般都处于移动网络环境下，网络带宽十分不稳定。主播端的码率变化必须足够灵活敏捷，能够快速响应无线网络中的带宽波动。同时，如果应用码率自适应技术，码率的平滑切换也是一个重要的指标。码率的平滑切换要求码率的切换幅度尽量小，切换次数尽量少；
- 直播的及时性。为了保证直播的高交互性和用户的高参与度，必须最小化主播端和用户之间的时延，或者限制住最大的时延[37]。端到端的时延要求反映在主播端，则对应于帧发送的时间和帧产生的时间差要满足一定的约束。

RTMP协议将视频帧作为基本的传输单位，码率的转换和调整都足够灵活，因此有可能能够满足上述三个性能要求。比如，它在主播端提供了大量可供调节的参数接口，可以调整视频质量，包括每秒的帧率，视频队列的大小，还有丢帧策略的相关参数。理论上来说，合适的丢帧策略需要在视频质量和及时性两者之间维持一个动态均衡[38][39][40][41]。直播情境下RTMP协议依然是主旋律。然而，在下章我们会给出，现在开源的RTMP实现以及商业级应用都存在严重的质量问题。

2.4　本章小结

本章首先以传统直播和交互直播的两大关键不同点作为切入点，个人直播相对于传统直播而言个人直播对传输技术提出了更高的挑战，因为个人直播的网络环境一般为无线网络，且用户和主播的强交互性对端到端的时延的要求较高。之后我们给出了现在的较为主流的直播传输框架，在主播端上传时使用RTMP协议，分发时使用HTTP协议借助CDN网络完成。

本章从三个角度来介绍直播传输协议的演变过程，直播框架，直播协议和自适应码率技术。传统直播一般使用P2P网络或者CDN网络来进行分发，还有用两种网络混合的架构来进行分发，交互直播架构的研究目前还只是停留在通过测量去发现现有直播的架构的地步。直播协议方面，RTMP协议因其较低的端到端时延而在直播应用中使用较多。HTTP协议因为至少一个视频块时长的时延而较少应用在直播中，但目前也有一些研究集中在减少HTTP协议的时延方面。码率自适应技术目前较多的研究集中在点播领域，直播方面目前的研究只涵盖了点对点的

直播，多用户情景的研究较少。

另外，本章给出了交互直播的三点性能要求，高码率，少切换以及端到端的时延。后续的几章内容主要通过测量发现直播系统中的问题，并尝试设计出一套相应的解决方案。

第3章　主播端的视频质量问题

我们用开源的直播软件和Nginx服务器搭建了一个实验平台，测量在无线网络环境下，直播应用能否有效的应对复杂的网络状况。通过测量，我们发现开源软件的主播端在运行过程中会遇到传输质量不佳的问题，尤其是当网络带宽发生抖动时。为了验证目前商业级的直播平台是否也存在类似的问题，本章我们又进一步测量了一些流行的个人直播应用，如斗鱼和Twitch。大量的测量表明商业级的直播平台也存在一样的问题。之后我们通过跟踪开源直播应用的源码实现，找出了问题发生的原因。

3.1　直播系统架构

传统直播要求主播端必须具有良好的专用带宽，而且传统直播系统的端到端的时延很大，为数十秒量级。为了解决传统直播的上述问题，交互直播应运而生。个人交互直播与传统直播相比，有两大关键不同点：

- 个人移动设备作为主播端。传统的直播，比如ESPN，都是使用预留的专线连接将摄像机捕捉的高清原始视频传输到特定的源内容服务器，内容服务器将原始的视频转码切分为视频块，分发给每个观看的用户。个人交互直播的不同点在于，用户借助移动设备通过无线网络上传直播视频到源服务器，源服务器分发的过程和传统直播相同。无线网络环境下，移动主播经常会面临和别人一起竞争带宽的情况，同时由于主播的移动和移动环境中复杂的无线信号，个人交互直播可能遭受更多变的网络环境，面临的传输挑战更加严峻；
- 移动直播要求主播和用户间的交互性。在传统的体育直播中，用户只是被动的观看直播，没有交互行为，也不需要得到反馈。交互直播由于存在主播和用户间的互动行为，端到端的时延至关重要，端到端的时延决定了主播和用户的交互体验。例如，如果用户给主播送礼物或者点赞，主播应尽快回复表示感谢，如果端到端的时延依然是传统直播的几十秒，用户的体验效果将会非常差。高交互性使得直播的延迟最多为几秒，远远小于传统直播的几十秒左右的端到端时延。

复杂多变的移动网络环境和高交互性的要求为优化交互直播的传输质量提出了很大的挑战。

图 3.1给出了个人交互直播的常用架构。交互直播的过程一般可以分为三个

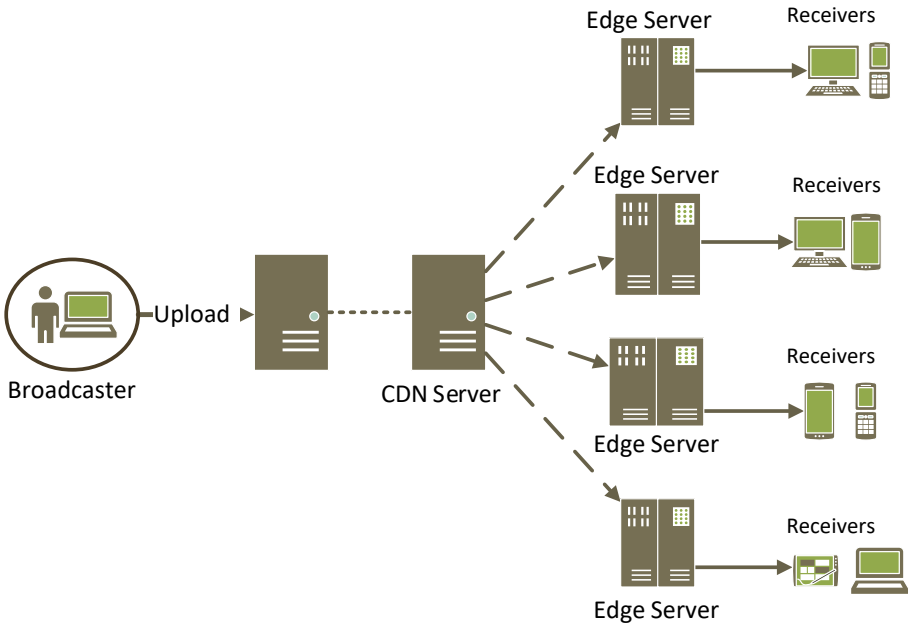


图 3.1　交互直播系统架构

部分，主播端到源服务器，源服务器到CDN边缘服务器，CDN边缘服务器到观众。开始直播时，一般主播端会通过特定的协议将直播的视频流上传到源服务器。主播端到源服务器端的协议多种多样，每个商业平台使用的协议都可能有所不同。多数的商业平台选择采用RTMP协议上传视频，少数的商业平台采用HTTP协议，也有部分平台使用自己定制的UDP协议去完成视频上传。源服务器在接收到主播端上传的直播视频流后，会首先将单一码率的视频流分别编码成多种码率。另外，如果通过CDN网络分发时使用的协议和主播端协议不一致，源服务器处会进行一定的转码操作。之后源服务器将编码后的视频流，选择合适的码率，转发至CDN分发网络；CDN网络将视频通过传统的overlay分发网络分发至CDN边缘服务器。最后，每个用户从最近的边缘服务器获取视频。CDN网络分发时使用的协议多数为HTTP协议，但也有少部分平台使用自己定制的UDP传输协议以减少端到端的时延。

3.2　主播端性能测量

实验设置 我们搭建的直播传输架构如图 3.2所示，包含两个主机和一个路由设备。两端的设备（主播端和观众端）都配备有两个CPU，6G内存，设备的网卡带宽为100Mbps，他们通过中间的交换机连接在一起。主播端配备有摄像头，通过OBS软件将摄像头抓取的内容上传至直播服务器。OBS是一个开源的商业级直播软件[42]，斗鱼和Twitch都将OBS作为推荐使用的直播软件。另外，OBS主要是使用RTMP协议来完成视频的上传。我们在主播端实现了一个带宽控制的模块，

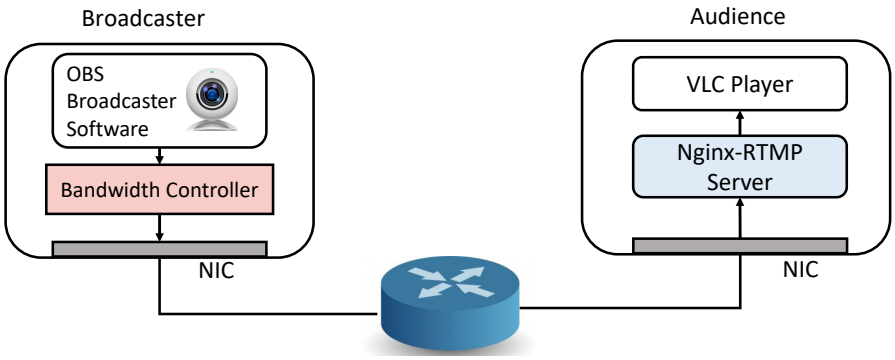


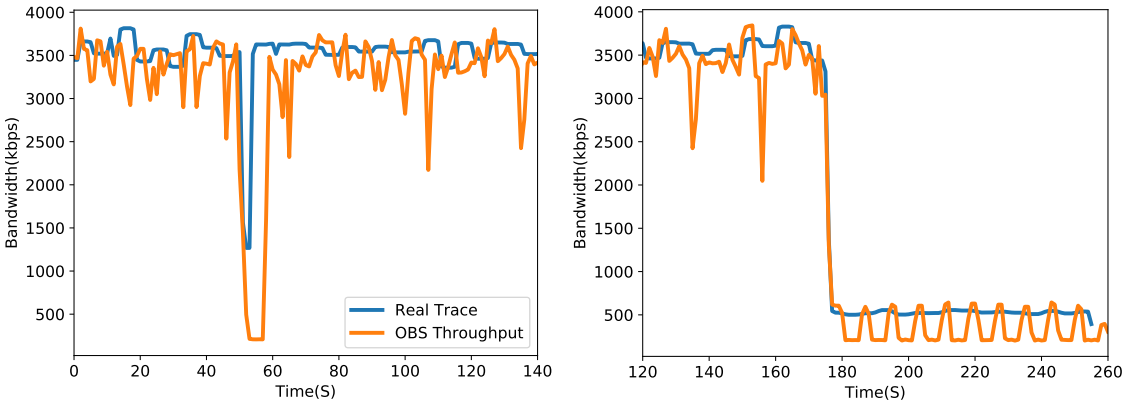
图 3.2　实验设置

能够实时模拟无线环境的网络带宽变化。带宽控制模块在Linux系统下用Linux系统自带的tc模块实现，在Windows系统下则用dummynet[43]来实现。在观众端，我们使用nginx的nginx-rtmp模块搭建了一个RTMP服务器，用来接收主播端上传的视频。另外，我们用VLC播放器去播放RTMP服务器接收到的直播视频流。

3.2.1　案例分析：带宽抖动导致传输质量差

为了尽量真实地模拟无线网络环境，我们使用真实的数据记录去控制实验的带宽变化。当用户通过移动设备浏览亚马逊网站时，每隔5秒给服务器发送探测到的实时带宽信息，我们以这种方式获取数据记录。我们将整个直播过程切分为多个5秒的时间段，将每个时间段内发送的所有数据报文聚合在一起，然后计算每个时间段内的总数据量，之后根据带宽的实际数值去控制主播端的网络带宽。我们选取其中260秒的记录作为具体案例，这一段数据的平均带宽在3300kbps以上。实验一开始我们把直播的码率设为3300kbps，低于平均码率，这是为了防止整个过程码率始终低于平均值，无法判断出直播平台应对网络抖动的能力。启动OBS，上传摄像头抓取到的视频；与此同时，在观众端启用tcpdump去实时抓取报文。整个过程实时的吞吐量结果如图 3.3所示，为了更清晰的展示实验结果，我们把数据记录分为两个部分，0-140秒以及120-260秒。

仔细观察上述实验图，我们可以得出以下两个结论。（1）从图 3.3(a)我们可以看出，大部分时间里，主播端的实际带宽一直紧紧跟随控制带宽发生变化。然而，在50秒附近，带宽大幅度衰减，低于初始的码率，维持在1500kbps左右，这种情况维持了2秒钟。此时，真实的数据吞吐量基本降为0，这一现象维持了8秒钟，从50秒到58秒。这是一种不正常的现象，2秒的网络抖动导致了直播流8秒的吞吐量下降。（2）从图 3.3(b)可以看出码率固定的策略并不能有效的应对长时间的带宽变化。0-180秒期间带宽始终是充足的，高于实时的码率3300kbps，但是180秒之后带宽急剧变化，下降到500kbps左右，这个过程持续了80秒的时间。在这期间

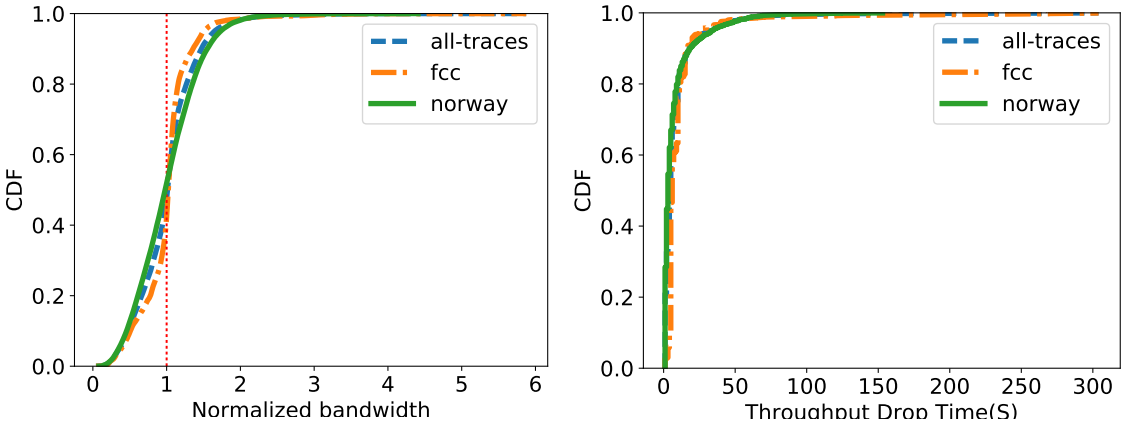


(a) 0-140s的吞吐量　　(b) 120-260s的吞吐量

图 3.3　无线网络环境下直播应用的吞吐量

主播端显示，发生了大量的丢帧。在网络很差的环境下，OBS主播端依然维持在默认的码率值，但实时带宽远远小于码率值，网络无法将产生的视频流推送出去，这个策略显然不能达到很好的效果。

3.2.2　带宽抖动的普遍性



(a) 归一化带宽累计分布函数　　(b) 带宽抖动时长分布

图 3.4　无线网络环境的带宽分布

为了评估无线网络环境下带宽抖动发生的频率，我们将两个真实的大规模公开数据集组合起来，以便数据分析。两个公开的数据集分别是Wifi网络环境下的FCC数据集[44]和移动网络环境下的HSDPA数据集[45]，其中每个数据记录的持续时长为320秒，两个数据集的总共持续时间为30多个小时。针对每一条数据记录，我们把平均带宽作为单位数，去归一化每个数据记录的所有数据点，总的cdf图如图 3.4(a)所示。有50%左右的时间，记录的数值小于平均带宽，这意味着对于一个10秒的带宽记录来说，有5秒的时间，实时带宽会低于平均值。20%的

情况下记录的数值只有平均值的一半。总的CDF分布图表明在真实网络世界中，带宽波动出现的很频繁。

为了进一步说明每次带宽抖动持续的时间，我们画了一张带宽抖动时长分布的CDF图 3.4(b)。带宽抖动时长是通过计算带宽在平均值以下的持续时间。大约20%的网络抖动其持续时长多于10秒，有些甚至持续数百秒。另外，我们在上面的两张图中还分别画出了FCC数据集和HSDPA数据集的带宽分布和抖动时间分布，两个数据集单独的分布和整体分布相同，差别不大。

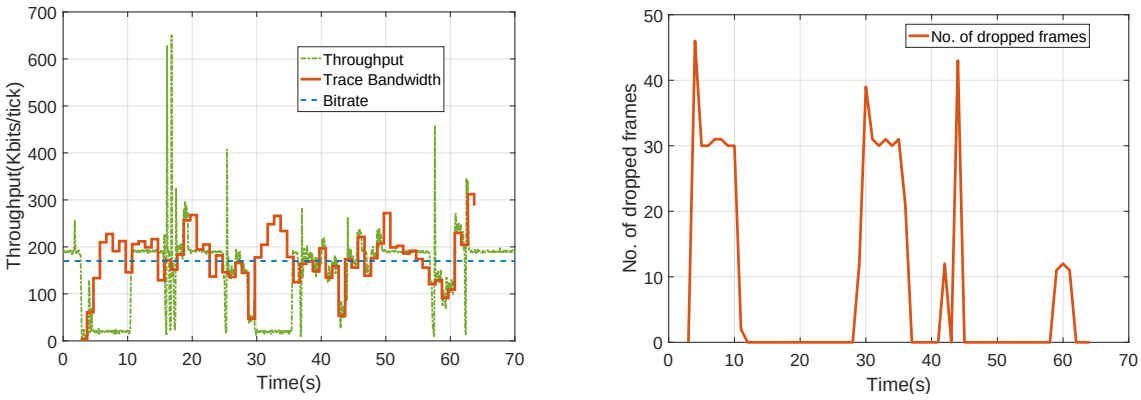
3.2.3　商业平台验证

经过对真实网络环境的分析，我们发现在无线网络状况下，带宽抖动发生的次数较为频繁，且每次带宽抖动都会持续一定的时长。通过对开源直播软件OBS的测量实验发现，带宽抖动情况下OBS会遇到视频质量下降的问题。为了验证流行的商业平台上是否存在相同的问题，我们选择了几个交互直播的商业平台去重复上述的实验。分别是OBS作为主播端推流到斗鱼服务器，斗鱼主播工具推流到斗鱼源服务器，OBS推流到Twitch的服务器。三组实验所用的带宽数据相同，初始的码率选择也相同，为1700kbps，数据记录的平均可用带宽高于初始码率。在实验过程中实时带宽偶尔会低于初始码率，我们用tcpdump去实时抓取真实的带宽。图 3.5(a)(b)(c)分别表示了三种实验环境下实际吞吐量的时序图，以及总丢帧数图。另外，表 3.1的前三行记录了每组实验的总丢帧数。

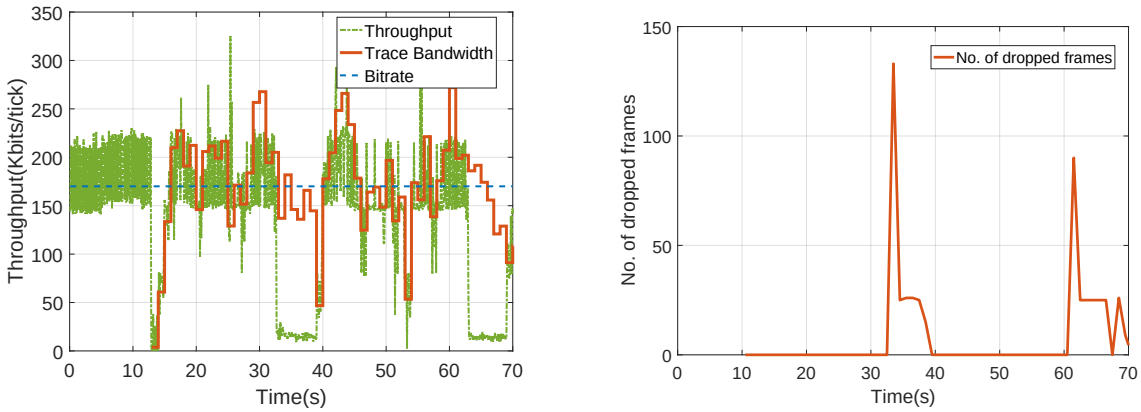
表 3.1　不同实验配置下的丢帧数

实验组	实验配置	上传失败时长(秒)	上传失败比例(%)
Figure 3.5	Obs主播端推流到斗鱼服务器	18.1	30.2%
	Obs主播端推流到twitch服务器	9.9	16.3%
	斗鱼主播工具推流到斗鱼服务器	16.6	27.2%
Figure 3.7	Obs主播端推流到斗鱼服务器	93.1	37.2%
	Obs主播端推流到twitch服务器	79.3	31.7%
	斗鱼主播工具推流到斗鱼服务器	66.67	26.7%

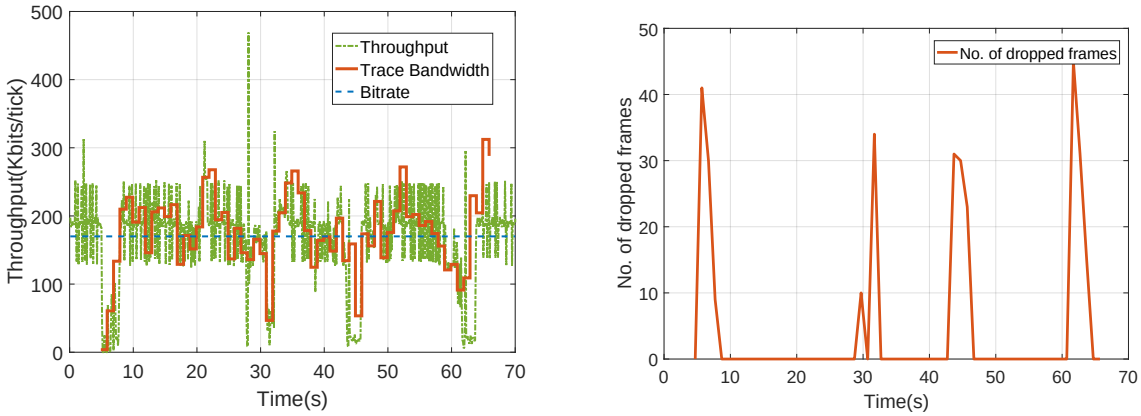
对比不同商业平台的测量结果，我们发现之前在OBS开源直播平台观察到的应用层上传暂停现象十分普遍，所有的商业平台都出现了类似的现象。大部分情况下，实际带宽是一直紧紧跟随着带宽的数据记录变化。OBS主播端推流至斗鱼服务器时卡顿现象出现在30-36秒间，斗鱼主播工具推流到斗鱼服务器的情况下卡顿发生在32-39秒，OBS推流端推流至Twitch服务器时卡顿现象发生在43-45秒。我



(a) OBS推流至斗鱼服务器的吞吐量和丢帧



(b) 斗鱼主播工具推流至斗鱼服务器的吞吐量和丢帧



(c) OBS推流至Twitch服务器的吞吐量和丢帧

图 3.5　商业平台验证实验

们观察三个时间段内的丢帧数，发现卡顿时的丢帧数一直维持在一个很高的数值。另外，我们从 3.5(a)(b)(c)三幅图中发现应用层卡顿放大现象的出现和带宽降低的幅度之间没有必然联系。例如，图 3.5(a)中，主播端的实时带宽在30秒时剧烈下降，降低为原来带宽的17%左右，此时出现了应用层放大效应。但在图 3.5(b)中，32秒时主播端实时的带宽轻微地下降，降低为原来的85%左右，轻微的带宽降低同样也触发了应用层的放大效应。由此可以得出一个结论，应用层放大效应的出

现并不取决于带宽降低的绝对幅度。另外一个发现是，应用层的卡顿放大现象导致的丢帧持续时长在不同商业平台上也各不相同：斗鱼平台上有多于5秒的丢帧，而Twitch平台上只发生了2-3秒的丢帧。表 3.1中显示，将斗鱼服务器作为RTMP源端服务器时，上传失败的比例在30%附近，Twitch服务器作为上传服务器时，上传失败的百分比只在16%附近，进一步验证了不同商业平台的丢帧时间不同。总之，上述的实验结果表明商业平台的直播端也不能够有效的解决短时间的带宽下降。

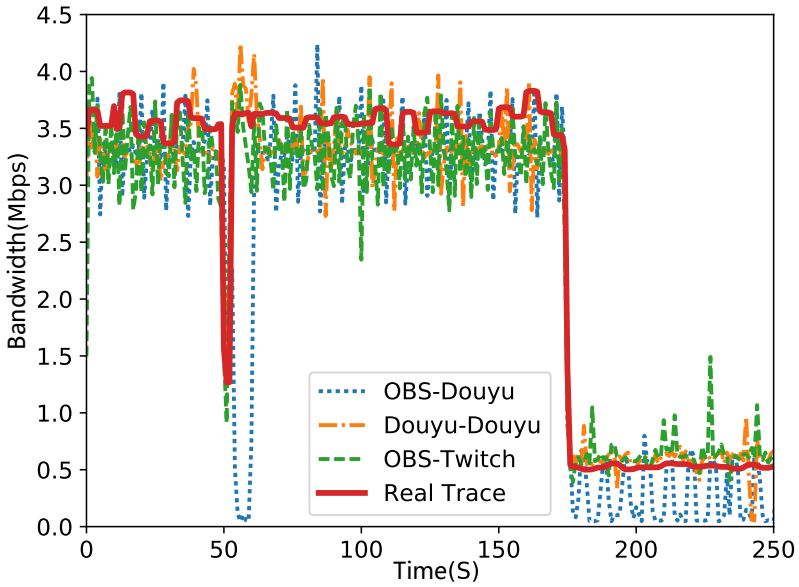


图 3.6　长时间带宽抖动状况下的吞吐量

除了短时间带宽下降的测量实验之外，我们还验证了在长时间带宽下降的情况时商业平台的性能表现，如图 3.6所示。与上面的实验配置大致相同，我们选取的初始码率3300kbps小于平均带宽，实验开始的一段时间里，实时的带宽一直高于初始的码率设置，主播端上传的视频播放很流畅。但在180秒之后时延带宽开始急剧的下降，降低到500kbps附近，实验期间我们一直持续观察码率变化，发现所有三组实验环境的码率始终保持不变，均维持在初始码率水平。但180秒之后，实时的码率大于可用带宽容量，视频数据产生的速度远远大于网络容量，观测到了大量的丢帧现象。OBS推流至斗鱼服务器的这组实验表现最差，甚至都没有充分利用带宽资源。尤其是180秒到250秒期间，这组实验的实时带宽在网络容量和零之间来回振荡。而观察其他两组实验，虽然整个过程中真实带宽紧跟网络容量变化，但180秒之后主播端一直在持续不断的丢帧。表 3.1的后面三行记录了总共的丢帧数据，三组实验的上传失败百分比都很高，进一步说明了现有的商业平台处理长时间带宽下降的能力非常的有限。总的来说，现有的商业平台不能有效解决长时间带宽下降的情境下丢帧的问题。

3.3　本质原因追溯

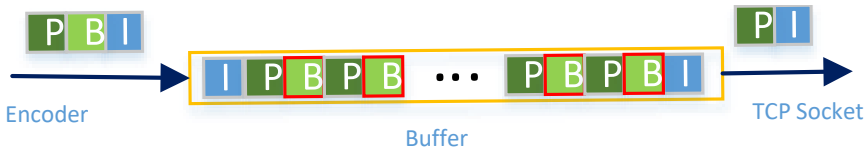


图 3.7　视频帧队列示意图

无论是有线网络环境还是无线网络环境，网络带宽抖动都时有发生。为了应对可能出现的带宽抖动，主播端应该会维护一个队列来存储等待发送的视频帧数据。主播端的视频帧队列的示意图如图 3.7所示。主播端通过摄像头实时抓取画面，之后将抓取的画面送入编码器，编码成H264格式的视频帧；编码完成后的视频帧被编码器释放出来，进入视频帧队列排队等待被发送。同时发送数据的进程从队列中顺序取出一个视频帧，将它通过TCP蹭的socket端口发送到网络中去，整个视频编码发送的过程便是这样。如果网络状况不好，网络容量低于视频数据产生的速率，数据发送的进程会被堵塞住，视频帧队列累积到一定程度会超出最大空间容量的限制，之后新产生的视频帧直接被丢弃，不再纳入帧队列，于是发生了丢帧现象。另外，视频帧队列里的数据帧也会因为超出最大容量的限制，而有可能会丢弃。

帧类型	I	B	B	P	B	B	P	B	B	I	...
播放顺序	1	2	3	4	5	6	7	8	9	10	...
编码顺序	1	3	4	2	6	7	5	9	10	8	...

图 3.8　H.264帧的编码顺序和播放顺序

丢帧现象产生的主要原因是网络堵塞。但大量的测量实验发现，丢帧可能还会在应用层造成放大效应。为了找到问题发生的根本原因，我们通过追溯开源直播软件OBS的源码，发现短时间的带宽抖动导致的丢帧放大效应主要是由于视频帧之间存在依赖关系。按照H264的编码标准，一段视频最后会被编码成一个图片组，称之为一个GoP（Group of Pictures）。编码协议规定，每个图片组的第一帧应保持原图不变，这一帧称为I帧，也称之为关键帧。P帧是利用前面已经编码的图像作为参考图像进行预测生成的，已经编码的图像包含前面的I帧和P帧；B帧则是一种双向预测，根据邻近的I帧和P帧产生。表 3.8给出了一系列连续的视频帧的示意图，按照播放顺序展示。表中可以看出，视频帧的编解码的顺序和最后实际播放的顺序并不一定一致。考虑到一个GoP里视频帧之间的依赖关系，当一

组GoP中间的P帧被丢弃的话，这个P帧之后所有的P帧和B帧都不能够正常解码。因此，如果网络带宽发生一次小抖动，导致一组GoP的中间或者开头出现丢帧现象，这个GoP剩余的帧都会被主播端丢弃，因此即使传输成功，也无法正常解码。这个原因导致我们在应用层观测到了放大效应，2秒的带宽抖动带来5-6秒的主播上传暂停。

Algorithm 1 OBS默认丢帧算法

Require: timespan：队列时间长度；dropPFrame：和P帧相关的丢帧优先级；dropBFrame：和B帧相关的丢帧优先级；bandwidth：每个时隙的带宽

```
T1 := 0.9秒,T2:=0.7秒,timespan :=0
if 新产生的帧是I帧 then
    dropPFrame := False, dropBFrame :=False
    入队(视频帧队列, 新产生的帧)
    timespan:= timespan + 1
if 新产生的帧是P帧 then
    if dropPFrame or timespan > T1 then
        丢弃(新产生的帧),丢弃所有的I帧和P帧
        timespane := timespan - 丢弃的时长
    else
        入队(视频帧队列, 新产生的帧)
        timespan := timespan + 1
if 新产生的帧是B帧 then
    if dropBFrame or timespan > T2 then
        丢弃(新产生的帧),丢弃所有的B帧
        timespan := timespan - 丢弃的时长
    else
        入队(视频帧队列,新产生的帧)
        timespan := timespan + 1
timespan := timespan - 发送的时长(bandwidth)
```

通过分析OBS主播软件的源码，我们破解了它的帧队列管理算法，如算法 1所示。OBS默认的丢帧算法引入了三个变量，P帧和B帧相关的丢帧优先级以及队列时间长度。丢帧优先级的设置是为了避免无效传输一些无法解码的视频帧，增加

对网络带宽的利用率；队列时间长度表示队列中最旧的帧和当前时间刻度之间的差值。当主播端发生丢帧时，丢帧优先级被置为True，视频队列停止接收GoP中剩余的其他帧。一开始，所有的丢帧优先级都会被初始化为False。当一个新的视频帧产生时，先判断该帧是否是I帧，如果是，加入视频帧队列，并将所有的丢帧优先级赋值为False。因为I帧标志着一个新的GoP的开始，所以所有的丢帧优先级都被赋为初始值，同时，将队列的时间长度增加一个视频帧的时长。如果新产生的视频帧是P帧，判断队列时间长度是否小于0.9秒，以及P帧对应的丢帧优先级是否为True。如果队列时间长度大于0.9秒，或者P帧对应的丢帧优先级为True，则丢弃当前帧，B帧和P帧的丢帧优先级都被设为True。而且如果时间长度多余0.9秒，丢弃队列里所有的P帧和B帧，时间长度减去队列中丢弃帧的时间长度。上述两个条件均不满足的话，将新产生的P帧加入视频队列，和I帧操作相同。如果新产生的帧是B帧，队列时间长度的阈值变为0.7s，其余的逻辑和P帧类似。

大量的测量表明，当直播过程中发生带宽抖动的情况时，主播端的码率依然维持在初始的码率，视频数据产生的码率远远大于可用的无线带宽容量，便会发生丢帧现象。如果丢帧时丢弃了一个GoP中位置靠前的P帧，那么为了避免传输无效视频帧，该GoP剩余的其他视频帧都将会丢弃，映射到应用层面的表现为一段时间的上传暂停，我们称之为应用层的放大效应。通过对于开源直播软件的分析，我们可以确定地说问题发生的根本原因就是由于H264编码标准使得帧与帧之间存在依赖性。

3.4　本章小结

本章通过在实验室搭建的demo平台上的测量发现，开源的直播软件OBS目前并不能有效的应对移动网络的带宽抖动。当短时的带宽抖动发生时，在OBS平台上观测到短时的抖动会在应用层引起丢帧的放大效应；而当长时间带宽下降出现时，OBS直播平台并不能有效的应对，视频帧产生的速度一直保持不变，远远超于网络容量，造成长时间的上传视频质量不高。

为了验证在现行的商业平台上是否存在相同的问题，我们选取了OBS和斗鱼主播工具作为主播端，斗鱼服务器和Twitch服务器作为接收端，去重复我们上述的实验。在商业平台上的测量结果和demo平台的结果基本一致，表明目前的商业平台也尚未能有效地应对移动网络的带宽抖动。通过解析OBS开源软件的源码，我们发现，应用层丢帧放大效应的主要原因是由于H264的编码标准导致帧与帧之间存在依赖性，长时间的带宽降低导致视频上传受到严重影响主要是由于视频的产生速率远远高于网络带宽容量。

为了解决上述问题，我们初步提出了两个方案，增加视频队列长度以及减少关键帧间隔。初步的仿真实验结果说明，两种方案都可以减少丢帧现象的发生，但是增加视频队列长度违反了帧传输及时性的要求。这为我们之后的优化空间提供了思路，视频队列长度应该严格控制在一个合理的阈值，将帧传输的时延控制在要求的范围内。

第4章　移动网络中主播端传输优化方案设计

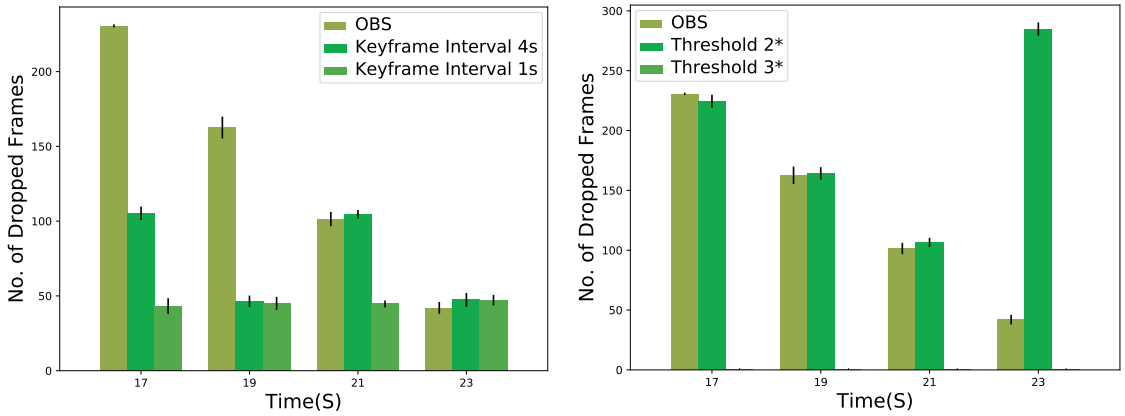
基于第三章大量的测量实验以及对于应用层丢帧放大效应的本质原因分析，我们重新思考将RTMP协议作为主播端协议时的方案设计，尝试从视频帧和GoP等三个不同的层面来定制化主播端的传输协议。首先，我们需要选择出一个合适的关键帧间隔，以减少视频帧之间的依赖性，从而缓解应用层放大效应的发生。然后，我们尝试最优化GoP内部的丢帧策略，去尽可能多地避免不必要的丢帧。前面提出的这两个解决方案协同作用，以应对短时间的带宽降低。最后，我们设计了一个GoP粒度的码率自适应算法，来缓解长期的带宽衰落所带来视频质量的下降。

4.1　优化空间

丢帧问题的发生明显地降低了观众的用户体验质量，应用层的丢帧放大效应加剧了上述问题。为了缓解丢帧问题，我们首先尝试了一些较为简单直接的方案，比如增加视频队列的长度，或者缩小视频的关键帧间隔。从理论上来说，这两个方案都可以减少丢帧现象的发生。因为我们知道帧之间的依赖性由于H264的视频压缩算法导致，非关键帧都是由关键帧计算得来。关键帧之间是相互独立的，依赖性只存在在一个GoP内部。通过减少关键帧间隔，我们可以减少视频帧之间的依赖性，从而减少丢帧。举个最极端的例子，如果关键帧间隔为1帧，那么帧之间完全不存在依赖性。另外，假设在关键帧为8秒的情况下，如果在一个GoP开始时，发生了一个2秒的网络抖动，那么整个GoP都会被丢弃；但是如果8秒的关键帧间隔被修改为2秒，就只会丢弃一个2秒的GoP。丢帧现象发生的另外一个原因是待发送的视频帧数据量超出队列最大容量的限制导致溢出，那么如果我们增加视频帧队列的长度，队列溢出的阈值被相应的提高，一定程度上可以减少一些视频丢帧。

为了验证上述提出的两种方案是否切实有效，并为我们的解决方案提供优化方向，我们对两种方案做一个初步的验证。

减少关键帧间隔 OBS直播软件关键帧间隔的默认值是8秒，实验中我们把关键帧间隔分别改为4秒和1秒。每次实验，我们均用OBS主播端去推流，同时为了控制变量，防止引入不必要的参数，OBS主播端推流时上传的是同一段视频。我们把OBS启动的时间定义为时间零点，为了观察关键帧对于丢帧的影响，我们分别在时间为17秒，19秒，21秒和23秒的时候断开网络连接，断开时长为2秒。丢



(a) 不同关键帧间隔下的丢帧数　　(b) 不同视频帧队列长度时的丢帧数

图 4.1　不同编码参数下的丢帧数

帧的结果图展示在图 4.1(a)中。我们可以看到，对于关键帧间隔相同的实验组来说，网络断开的时间位于一组GOP越早的时间处，丢弃的帧数则会越多。因为如果是GOP组中较早的帧，会有更多依赖于它的帧。对于默认为8秒的关键帧间隔，当断开时间分别是17秒，19秒，21秒和23秒时，丢帧数分别为238,164,105和48。因为对于关键帧间隔为8秒的情况，一个GoP的时间为16到24秒之间，17秒位于一个GoP的第1秒处，23秒的帧位于一个GoP的结尾1秒处，观察到的丢帧数也依次递增，验证了之前的理论猜想。

另外，对比在17秒时网络断开的实验组，我们发现关键帧间隔越小丢帧数减少地越明显，比如，当关键帧从8秒变到4秒再变到1秒时，丢帧数从238减到102，最后减少到46。但对于在23秒断开网络连接的实验组来说，丢帧数减少的效果并不明显。这是因为23秒处的帧在三组实验中均接近于GOP的尾部，都只有1秒时间的帧依赖于23秒处的帧。

增加视频队列长度 OBS直播软件中默认的队列长度为，P帧0.9秒，B帧0.7秒。控制实验中我们尝试把队列长度分别改变为原来数值的2倍或3倍。原来的队列长度组合为（0.7,0.9）秒，变为2倍和3倍之后，队列阈值组合分别为（1.4,1.8）秒以及（2.1,2.7）秒。和减少关键帧间隔的实验类似，我们重复上面两组推流过程，同时记录下来每次实验的丢帧数，最后的统计结果展示在图 4.1(b)中。当队列长度增加为原来的2倍时，丢帧数并没有减少，在17秒，19秒和21秒的情况下，丢帧数基本保持不变。另外，图中可以看出在23秒时断开网络连接，丢帧数反而呈现增加的趋势。这是因为网络断开时间为2秒钟，在23秒时断开网络，队列里会缓存着一个GoP的最后一秒视频帧以及下一个GoP的开始1秒视频帧。而且OBS的丢帧策略是一旦发生丢帧，丢弃视频队列里的所有P帧和B帧，则丢帧行为会丢弃一整组GoP加一秒的视频。但当我们把队列长度增加为3倍时，丢帧数基本减少为零，

这是因为队列长度的丢帧阈值为2.9秒，远远大于网络断开的时间2秒，丢帧数因此均为0。

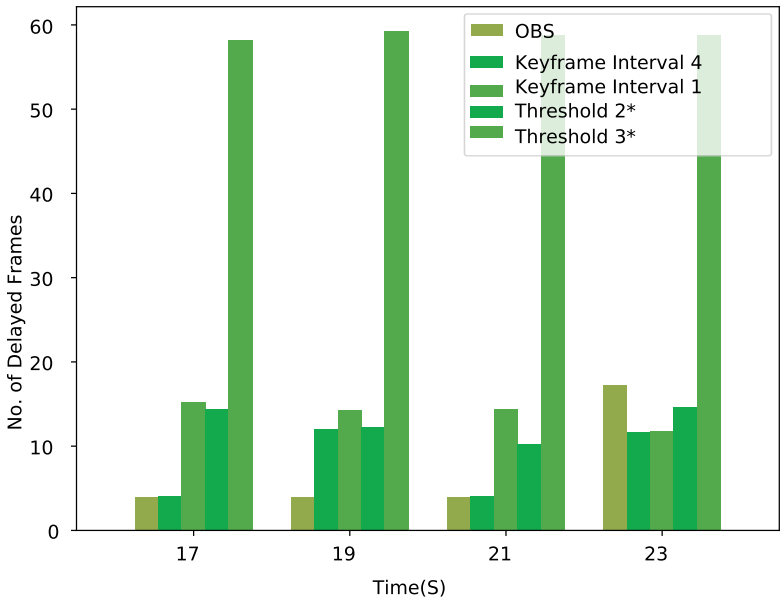


图 4.2　延迟帧的总数

及时性 除了将总丢帧数作为我们的衡量指标之外，每个视频帧是否被及时的发送出去也是我们的一个重要衡量指标。为了衡量上述几组实验的及时性，我们在RTMP接收服务器处设置记录点，记录下每个视频帧的编码时间和到达时间。我们把第一个视频帧的到达时间和编码时间的差设置为0，为初始时间，记录其余视频帧和上一视频帧的编码时间和到达时间的差值，若到达时间晚于编码时间，则该视频帧被延迟了。统计总共延迟的视频帧数，并记录在图 4.2中。从图中可以看出，当关键帧间隔变为4秒和1秒时，延迟的帧数目变化不大，维持在10帧以内，改变关键帧间隔对于及时性的影响很小。但将视频帧队列的长度增加为原来的3倍时，延迟的帧总数变为60帧左右，增加视频帧队列的长度会严重影响视频帧发送的及时性。

实验结果表明，当减少视频的关键帧间隔之后，短时间的网络带宽抖动只能引发一个相对较短时间的丢帧效应，并没有在应用层引发进一步的放大效应。增加视频帧队列的长度也可以减少丢帧的数量。这是因为增加视频队列的长度相当于提高了丢帧现象发生的阈值，从而可以减少丢帧现象的发生，因此减少一部分的丢帧。

及时性指标在直播的用户体验中也占据着重要的地位，但是上面两种方案中，增加视频队列的长度违反了及时性的要求。因此，为了满足及时性的要求，我们在之后提出的解决方案中，将视频队列长度限制在足够小的数值。之后的实验中，

我们将视频队列的空间大小限制为0.9秒。另外，前面的初步验证给了我们改善视频传输质量的一些方向，比如，减少关键帧间隔。结合对于丢帧根本原因的分析，我们提出通过优化视频帧产生和传输的机制去适应变化的无线网络环境，改善用户体验质量，主要包括下面几点：

1) **减小视频帧之间的依赖性** 帧之间的依赖性是由于H264的视频压缩算法导致的，通过减少关键帧间隔，帧和帧之间的依赖被弱化了。然而，减少关键帧间隔会损失一定的视频质量，这个方法需要在丢帧数量和视频质量之间达到一个平衡。因为一组GoP中，一个I帧所占用的字节数很多，远远大于P帧和B帧，减少关键帧间隔，意味着同样数量的视频帧中I帧的数量增多。在同样码率的情况下，I帧数量的增多意味着每一帧分配到的数据量减少，编码器加强每一帧画面的压缩，画面的质量将会有所降低，甚至出现大的像素块；
2) **智能的丢帧策略** OBS现在的默认策略是当队列时间长度超出规定的阈值后丢弃视频帧队列里所有的P帧和B帧。目前的丢帧策略存在一定的合理性，因为如果丢弃视频帧队列中位于队列头部的视频帧，和它属于同一个GoP中剩余的其他帧也就无法解码。但是反之，如果丢弃最新的视频帧，时间较为久远的视频帧依然存在在队列中，及时性就无法保证。但是，对于队列中同时存在2个或者多个GoP的情况，一个直接的可以优化的点就是，丢弃旧的GoP的帧。这样的方法相对于OBS的默认策略会有一定的提升。设计出一个接近最优解决方案的在线的丢帧策略很重要，唯一的挑战在于帧之间存在依赖性。暴力搜索方式虽然可以找出最优解，但是由于时间复杂度太高，所以实际中并不可用；
3) **自适应码率** 图 3.4说明了网络带宽抖动频繁发生，第三章的测量结果显示商业平台只使用固定码率或者平均码率的方式去编码视频。平均码率的方式是指实际的带宽可以在目标码率附近波动。这两种编码方式都不能动态跟随带宽变化，当带宽下降时，会发生大量的丢帧。如果带宽降低持续时长较长的话，这种现象尤其明显。一个可能的解决方案是类似于点播情况下的DASH，在主播端动态改变码率。我们的选择是以GoP的级别去改变码率，为每一个GoP选择一个码率。引入码率自适应的方式可能会大幅度的减少丢帧。

4.2　GoP的最优选择

如果关键帧间隔过大，当网络带宽有瞬时的降低时，会在应用层产生放大效

表 4.1　丢帧策略模型所用符号表

符号	类型	含义
i	编号	帧编号
j	编号	时间戳编号
x_{ij}	变量	第j个时刻，第i帧是否在视频帧队列中
y_{ij}	变量	第j个时刻，第i帧是否已经被发送
z_{ij}	变量	第j个时刻，第i帧是否已经被丢掉
T	常量	总的决策时长
T_1	常量	一个帧能够在视频帧队列中留存的最大时长
C_j	常量	第j个时刻的网络带宽
N	常量	两个关键帧间的帧间隔
S	常量	每一帧的数据量大小
M_j	常量	第j个时刻可以发送的最大帧编号

应，丢帧数较多。但如果关键帧间隔设置的过小也会导致过高的视频压缩比，视频的清晰度会受到损害。因此，最优的GoP选择需要权衡丢帧数和视频质量两个因素。

关键帧间隔的大小对于丢帧数的影响可以通过精确的控制实验去衡量，如图 4.1(a)所示。视频质量方面，我们用SSIM作为指标去衡量视频的清晰度[46][47]，SSIM通过对比两幅画面的相似度去衡量视频质量。在H264编码中，SSIM旨在计算一副图片编码前后的相似性。我们重复进行多次试验，每次实验均使用同一视频，改变关键帧间隔的大小，找出SSIM在最高值[95% – 100%]范围内的最小关键帧间隔。

4.3　智能丢帧策略

除了选择最优的关键帧间隔，针对短时间的网络带宽下降，一个好的丢帧策略也可以达到不错的效果。好的丢帧策略可以在丢帧数量和视频质量之间达到一个动态的平衡[41]。我们先假设已知全程的网络带宽，通过数学建模分析尝试找出理论上最优的丢帧策略。理论最优的算法时间复杂度较高，我们设计了一个低时间复杂度的在线算法，低时间复杂度的算法对移动设备计算能力的要求没那么高，且在实际中可以真实部署。

4.3.1　丢帧策略建模

首先，我们为了从理论上给出最优的丢帧策略可以达到的最好用户体验质量，对丢帧问题进行了建模。假设每组GoP的视频帧格式已经确定，整个过程的网络

带宽状况已知，一定存在一个最优的丢帧策略，在满足带宽和队列时间长度限制的情况下能够最大化观众端的体验质量。每组GoP图片组包含三种类型的帧：I帧，P帧和B帧。为了简便，建模时我们暂时忽略B帧，去研究丢帧策略的本质。所有的符号都定义在表 4.1中，我们将整个决策过程均匀地分为T个时隙，第i个时隙产生的帧标号为i。x_{ij}，y_{ij}，z_{ij}都是01变量，分别表示在第j个时刻第i个帧是否在队列中，还是已经被发送出去或者丢弃。

帧留存性约束 主播端每个时刻均会产生一个视频帧，第i个时刻新产生的视频帧标号为i。第i个时刻之后，第i个帧有三种去向，要么保留在视频帧队列中，要么被发送到网络中，或者被主播端丢弃。视频帧的去向可以用如下的表达式(4-1)(4-2)描述：

$$x_{ij} + y_{ij} + z_{ij} = 0, \forall j < i \tag{4-1}$$

$$x_{ij} + y_{ij} + z_{ij} = 1, \forall j \geq i \tag{4-2}$$

如果一个视频帧已经从视频帧队列中移除，那么之后永远都不会再进入队列。同样的，一个帧被发送到网络中或者被丢弃之后，它的状态就永远变成了已发送或者被丢弃。如式（4-3）（4-4）（4-5）所示：

$$x_{ij} \geq x_{i,j+1}, \forall j \geq i \tag{4-3}$$

$$y_{ij} \leq y_{i,j+1}, \forall j \geq i \tag{4-4}$$

$$z_{ij} \leq z_{i,j+1}, \forall j \geq i \tag{4-5}$$

带宽约束 每个时刻应该发送哪些视频帧也是减少丢帧的一个决策空间，即最优的发送策略y_{ij}，也是一个重要且有趣的问题。然而，我们关注于求解最优的丢帧策略，这里我们先假设主播端在满足网络容量约束的条件下，每次尽可能多的发送内容。假设每个时隙最大可发送的帧编号为M_j，那么第i个时刻y_{ij}的转移方程（4-6）如下：

$$y_{ij} = x_{i,j-1}, \forall j, i \leq M_j \tag{4-6}$$

等式除了满足M_j的约束外，可以被发送的帧必须在队列里。满足这些约束，每个时隙最大的可发送的帧编号M_j可以通过下面的等式（4-7）来计算：

$$M_j = argmax\Sigma_k x_{k,j-1} \leq C_j \tag{4-7}$$

及时性约束 一个帧在产生后的$T1$秒内必须发送出去或者被丢弃，否则就会违反了及时性约束条件。所以说，一个帧在产生的$T1$秒后，要么是被发送出去，要么就是被丢弃，如式（4-8）。

$$y_{ij} + z_{ij} = 1, \forall j > i + T_1 \tag{4-8}$$

解码约束 发送到网络中的帧必须要满足解码条件，否则即时视频帧成功发送到接收端也无法解码，这是对带宽资源的一种浪费，并没有充分的利用带宽。根据H264的编码准则，I帧始终可以解码，P帧只有在前面的I帧和P帧都接收到的情况下才能正常解码，如式（4-9）所示。

$$y_{i+1,T} \geq y_{iT}, \forall i \not\equiv N - 1(\mathrm{mod}N) \tag{4-9}$$

优化目标 智能丢帧问题，其最终的目标函数是去最大化传输成功的可解码视频帧数。与视频点播场景下的传输问题相比，我们建立的模型考虑了帧的传输及时性以及接收端是否能正常解码等约束条件，更加符合个人交互直播的场景。具体的目标函数如式（4-10）下：

$$\min \Sigma_i y_{iT} \tag{4-10}$$

4.3.2　启发式算法

分析上述目标函数和所有的约束条件可知，需要求解的变量为x_{ij}，y_{ij}，z_{ij}，分别代表在第i个时隙第j个视频帧的去向，保留在视频帧队列中，发送到网络中，或者被主播端丢弃。这些变量的取值只有0和1两种，因此这是个整数规划问题。如果已知全程的带宽状况，自然可以求得该问题的离线最优解法，然而实际情况中我们并不能提前已知全部的带宽状况。另外，离线最优解法一般是通过暴力搜索所有的取值可能性去求得，该算法的时间复杂度较高，占用设备资源也较多，对于移动设备来说难以承受。因此，直接求解整数规划的解法在实际情况中不可用，急需一个在线的丢帧策略。

考虑到两个或者多个GoP同时出现在视频帧队列中的情况，我们提出了改进版的丢帧算法，GreedyDrop，如算法 2所示。不同于默认算法将视频队列中所有的P帧都丢弃，GreedyDrop仅仅丢掉旧的GoP中的所有P帧，具体见算法的第8行描述。因此新产生的GoP被保留了下来，在这种情况下，我们的算法至少减少了一个GoP数量的帧丢失。

Algorithm 2 GreedyDrop丢帧算法

Require: timespan：队列时间长度；dropPFrame：和P帧相关的丢帧优先级；bandwidth：每个时隙的带宽

```
T1 := 0.9秒, timespan :=0
if 新产生的帧是I帧 then
    dropPFrame := False
    入队(视频帧队列, 新产生的帧)
    timespan:= timespan + 1
if 新产生的帧是P帧 then
    if dropPFrame or timespan > T1 then
        丢弃(新产生的帧),丢弃队列中较旧的GoP的I帧和P帧
        timespane := timespan - 丢弃的时长
    else
        入队(视频帧队列, 新产生的帧)
        timespan := timespan + 1
timespan := timespan - 发送的时长(bandwidth)
```

4.4　自适应码率

上述提出的两个方案旨在解决短时间带宽抖动带来的应用层放大效应。为了进一步解决长时间的带宽抖动所带来的质量下降问题，我们尝试引入动态自适应码率的方式。

4.4.1　动态码率建模

和固定码率的算法相比，动态码率情况下最优的丢帧策略也会有所不同。但因为我们的研究重点是如何平滑高效的动态变化码率，这里我们先采用上一小节提出的智能丢帧策略GreedyDrop。首先，我们还是先尝试计算出通过调整码率能达到的最佳视频质量；之后，为移动设备设计一个在线可用的简单算法。

建模过程使用都得符号全部包含记录表 4.2中。为了研究最优的动态码率策略，我们引入了一组和码率相关的变量，其中R_j表现第j帧所选取的码率。使用GreedyDrop作为我们的丢帧策略，最大化用户端的视频传输质量可以建模成下面的式子（4-11）。

$$\max \sum R_j - \alpha \sum |R_{j+1} - R_j| - \beta \sum D_j \tag{4-11}$$

表 4.2　动态码率符号表

符号	类型	含义
j	编号	帧标号
R_j	变量	第j帧的码率
N_j	变量	第j个时刻的GoP组数
D_j	变量	第j个时刻的丢帧数
S_j	变量	第j个时刻发送出的GoP组数
C_j	变量	第j个时刻的网络带宽
T_k^j	变量	第j时刻第k个GoP的剩余时间
R_k^j	变量	第j个时刻第k个GoP的码率
$Rest_j$	变量	第j个时刻发送剩余的GoP的时长
T	变量	总决策时长
T_1	变量	一个帧能够在队列中留存的时间阈值

等式中的第一项R_j是第j帧时选择的视频清晰度所带来的收益，第二项$|R_{j+1}-R_j|$代表由于连续的两个视频帧码率切换带来的效益损失，最后一项D_j表示第j个时隙由于网络抖动导致丢帧带来的效应损失。变量α和β对应码率切换损失和丢帧损失的系数。

码率约束 因为我们的解决方案中码率调整的粒度为一个GoP组，所以一个GoP内部的码率必须相同，如式（4-12），其中，mod是取余函数。

$$R_{j+1} = R_j, \forall mod(j, M) \neq M - 1 \tag{4-12}$$

带宽约束 在有限带宽的情况下，每个时隙最多能够发送出的GoP组数定义为S_j，其具体的计算公式如式（4-13）：

$$S_j = argmax \sum_k R_k^j * T_k^j \leq C_j, \forall j \tag{4-13}$$

及时性约束 视频队列的时间长度应时刻小于规定的时间阈值，否则，执行丢帧操作。第j个时刻在所有能够发送的视频帧被发送出去之后，视频队列里剩余的时间长度记为$Rest_j$。首先需要判断剩余的时间长度$Rest_j$是否大于时间阈值$T1$，如果$Rest_j$大于$T1$，则丢弃队列中时间较为久远的一组GoP。上述行为可以用下面的等式（4-14）(4-15)(4-16)来表示：

$$Rest_j = (C_j - \sum_{S_j} R_k^j * T_k^j)/R_{S_j+1}^j, \forall j \tag{4-14}$$

$$F_j = sgn(\sum_{S_j+1} T_k^j - Rest_j - T_1), \forall j \tag{4-15}$$

$$D_j = F_j * (T_{S_j+1}^j - Rest_j), \forall j \tag{4-16}$$

其中，sgn是修正版的符号函数，当变量大于0时，函数值等于1；否则函数值等于0。

状态转移方程 等式（4-17）描述了第j+1个时隙的GoP组数，等式（4-17）的最后两项 $1 - sg(mod(j, M))$ 取决于第j个帧是否是关键帧。约束条件组 (4-18) (4-19)(4-20) (4-21) (4-22)是队列中现存的每组GoP的码率和剩余时间的转移方程。码率和剩余时间的转移方程都需要考虑到当前帧是否是关键帧的情况，因为如果当前帧是关键帧，这代表着一个新的GoP的开始。等式(4-19)和(4-22)便是针对第j帧是关键帧考虑的特殊情况，约束条件为$mod(j, M) = 0$。除此之外，剩余时间的计算还需考虑丢帧行为带来的影响。约束(4-21)则给出了存在丢帧情况时时间最为久远的GoP剩余时间转移方程。

$$N_{j+1} = N_j - S_j - F_j + 1 - sgn(mod(j, M)), \forall j \tag{4-17}$$

$$R_k^{j+1} = R_{k+S_j+F_j}^j, \forall j, k \in \{1, N_j - S_j - F_j\} \tag{4-18}$$

$$R_{N_j-S_j-F_j+1}^{j+1} = R_{j+1}, \forall mod(j, M) \equiv 0 \tag{4-19}$$

$$T_k^{j+1} = T_{k+S_j+F_j}^j, \forall j, k \in \{1, N_j - S_j - F_j\} \tag{4-20}$$

$$T_{N_j-S_j-F_j}^{j+1} = T_{N_j-S_j-F_j}^{j+1} - D_j - Rest_j, \forall j \tag{4-21}$$

$$T_{N_j-S_j-F_j+1}^{j+1} = 1, \forall mod(j, M) \equiv 0 \tag{4-22}$$

离线最优的解决方案很难求解。假设对于每个GoP组，主播端可以从M个码率中任意选择，那么对于T个GoP组，码率选择的计算复杂度等于M^T，达到了指数级复杂度。

4.4.2　启发式动态码率算法

指数级复杂度的问题很难在有限的时间内求得答案。除此之外，离线最优算法需要已知未来的全部带宽。这种长时间的带宽预测精确度很难保证，一个直接的想法是根据实时带宽改变视频的码率。视频队列中的剩余数据量大小也是选择码率的有效依据。我们提出了启发式的动态码率算法，简称GVBR，算法的伪代码如图 3所示。根据GVBR算法，在第j个时刻，主播端执行下面两个关键步骤。η是有关丢帧和码率的系数。

Algorithm 3 GVBR码率自适应算法

1: Rest:=0, Send:=0, Drop:=0, η
2: **for** j = 1 to T **do**
3: 根据前面tau个时隙的历史带宽记录[C_{j-tau}，C_{j-1}]，计算调和平均值来预估第j个时隙的带宽C_j
4: 选择一个最大可用码率R_j，小于等式$(C_j - rest)/\eta$
5: 在满足带宽约束的条件下，发送队列中的视频帧，发送的帧数为 $Send$
6: 判断是否需要丢弃额外的帧，丢帧数为 $Drop$
7: 计算视频帧队列中剩余的数据量 $Rest = Rest + R_j - Send - Drop$

1) 带宽估计 在Festive和MPC两篇码率自适应研究论文中，均使用调和平均值来估计未来时隙的带宽。显然码率估计的精确度越高，我们算法的性能会越好。这篇文章中，因为我们的研究重点不是对带宽进行预估，我们暂时采用调和平均的方式去估计未来几个时隙的带宽。
2) 码率选择 为了避免在直播过程中由于网络抖动而导致频繁的丢包，一个合适的码率选择算法尤为重要。假设已知未来的带宽C_j和队列现在剩余的数据量大小$Rest$，我们提出的启发式算法会选择比$(C_j - Rest)/\eta$低的最高码率。这是因为未来可用的带宽减少等待传输的数据才是真实的网络容量。η参数的设置是为了纠正对于真实网络容量的预估。

GVBR结合了我们提出的三种策略，是一整套改善主播端传输质量的解决方案，包含GoP层面和帧层面。

4.5 本章小结

本章从三个层面提出了针对移动网络状况下主播端的传输质量优化机制，包含GoP层面和GoP内部的机制。

首先，我们通过多次重复控制实验的方法来确定最优的关键帧间隔。最优的关键帧间隔选择要同时考虑丢帧现象的发生以及视频质量的影响。

其次，本章对于移动网络状况下主播端的丢帧策略进行了理论建模，考虑视频GoP内部帧结构以及网络带宽全部已知的情况下，最优的丢帧策略。同时，考虑到移动设备的计算资源以及实际运行的情况，提出了一个在线的启发式丢帧策略，命名为GreedyDrop算法。

最后，在前两步优化的基础上，进一步优化由于长时间的网络带宽抖动带来的视频质量下降的问题。我们对移动网络状况下主播端的码率自适应算法进

行了理论建模，丢帧策略和关键帧间隔采用前两步的结论。动态码率的效益函数综合考虑了每个GoP选择的码率，码率间的切换以及丢帧数等因素。为了实际可用，我们根据视频帧队列的剩余空间以及网络带宽去选择当前的码率，命名为GVBR算法。GVBR算法结合了三种算法的优势，是一整套的移动网络下主播端传输优化的解决方案。

第5章　算法实现和性能评估

本章节主要评估了前一章节三种策略的性能优化效果，包括减少关键帧间隔，智能丢帧策略GreedyDrop，以及最后的整套方案，GVBR算法。在本文所有的仿真实验中，我们将帧率都设置为30帧每秒。

5.1　最佳GOP

4.1章节的初步实验验证表明减小关键帧间隔可以降低丢帧。但减少关键帧间隔实际上是个很棘手的选择，因为关键帧数目的增加可能会导致视频质量的降低。在本节中，我们尝试去具体评估由于减少关键帧间隔所带来的性能变化。

视频质量和关键帧　关键帧间隔的选择需要在视频质量和丢帧数之间达到一个动态均衡。为了指导关键帧间隔的选择，我们重复进行了多次实验，每次实验改变关键帧间隔大小，将未压缩的视频流压缩成H264格式，测量压缩后的视频画面质量。实验中使用x264编码器[48][49]去编码原始视频，实验中需要进行编码的原始视频为同一视频片段。x264编码器工作在差分编码模式，两个关键帧之间相隔的帧越多越有可能引起累计误差，GoP的推荐值是小于250帧。虽然最大限制值为250帧，但具体的关键帧间隔取值依然很复杂。我们使用的视频数据集包含各种不同类型的视频，超清视频，高清视频，游戏和4k视频[50]。关键帧间隔大小和归一化SSIM的关系展示在图 5.1中，SSIM的单位值为一次实验中最大的SSIM值。我们从众多视频集中挑选出四组视频，代表整体的实验结果。从图中可以看出，当关键帧间隔大于0.5秒后，由于编码器压缩量化导致的视频质量损失很微小，SSIM的数值基本保持不变，在最大SSIM的(97%-100%)范围内。结合前面对关键帧间隔和丢帧数的关系的测量，选择小的关键帧间隔会减少丢帧数。总结上面的实验，我们给出了关键帧的建议选择范围[0.5, 2]秒。

5.2　智能丢帧策略

5.2.1　智能丢帧策略的实现

为了衡量丢帧算法的性能优化效果，我们选择了两个算法作为对比，离线最优算法Oracle和默认算法OBS。Oracle，通过暴力搜索加剪枝可以达到离线最优，但其时间复杂度在指数级别。我们从带宽数据集中任意选择其中一段，时长为30秒附近。利用上一小节的结论，关键帧间隔的选择范围为[0.5, 2]秒，我们将

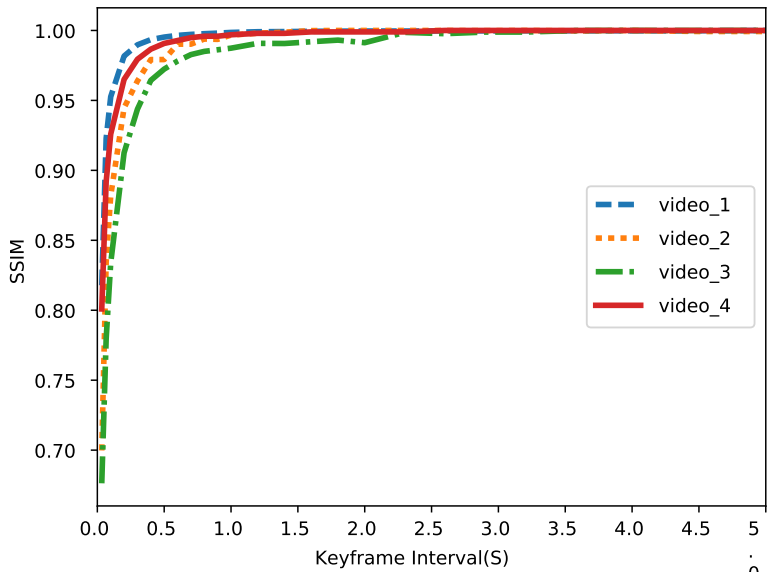


图 5.1　不同关键帧间隔时视频编码后的SSIM

关键帧间隔设置为1秒。为了研究算法优化丢帧的效果，在三组实验中我们将码率都固定在2850kbps，小于初始带宽值，但远大于衰落之后的带宽值。合理的码率选择使得这段数据既包含长时间的带宽抖动，也包括瞬时的带宽抖动。视频的帧率依然选择为30帧每秒。

GreedyDrop算法的具体实现见流程图 5.2。每当一个新的视频帧产生时，会启用图中的机制去处理视频帧。首先，判断和P帧相关的丢帧优先级是否被置为$True$，如果被置为$True$，则丢弃视频帧队列中时间较为久远的视频帧，直到碰到I帧时才会停止丢帧，将队列时间跨度在原有值的基础上减去丢帧时长。另外，如果视频帧队列中存在关键I帧的话，则将丢帧优先级置为$False$。其次，下面的步骤则是在满足带宽约束的条件下尽可能多的发送视频帧，将队列时间跨度在原有值的基础上减去已发送的时长。最后，判断现在的时间跨度是否大于阈值$T1$，如果大于P帧的阈值，则丢弃视频队列中所有不满足及时性约束的帧，同时将队列的时间跨度减去丢弃的时长，将丢帧优先级置为$True$。一个视频帧的完整处理机制就是这样，完成对于视频帧的处理后便等待着一个新的视频帧产生，以触发上述机制。

离线最优丢帧算法Oracle的核心代码部分如图 5.3所示。对于每个时隙来说，可能的丢帧策略数和视频帧队列中GoP的组数有着相关关系。对于视频帧队列中的帧排列，丢帧首先需要满足可解码性的要求，即丢帧之后队列中剩余的视频帧都必须可以正常解码。所以一个合理的丢帧策略是，对于视频帧队列中的每一个GoP，丢弃每个GoP组结尾连续的视频帧。Oracle算法的核心代码部分用递归的想法来实现，对于每个GoP，列出所有可能的丢帧组合，并继续往下递归直到循

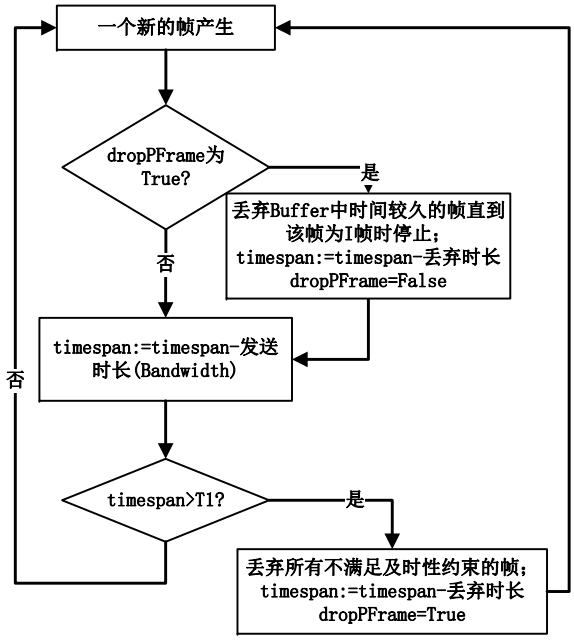


图 5.2　GreedyDrop算法的实现流程图

表 5.1　与默认OBS算法相比丢帧的减少程度

算法	丢帧数	上传失败时长（秒）	与默认OBS算法相比丢帧对应的百分比
Oracle	265	8.83	80%
GreedyDrop	274	9.13	85.6%
OBS默认算法	320	10.67	无

环的GoP组数等于队列包含的GoP组数。具体例子如下所示，例如，对于一个含有k个帧的GoP来说，合理的丢帧策略总共有k种，即，丢弃0个帧，丢弃最后一个帧，丢弃最后两个帧...丢弃最后k-1个帧和丢弃所有的帧。对于每一个GoP来说，从k种情况中选择一种，继续循环下一个GoP的情况，所以对于一个有N个GoP的视频帧队列来说，丢帧情况的组合数为指数次复杂度。由于Oracle算法的复杂度过高，因此在仿真实验时，我们只取了30秒时隙的数据集用来仿真。

5.2.2　丢帧策略的效果评估

在该数据集下，三种算法丢帧的减少程度在表 5.1中给出。本文中，我们定义了一个新的指标来衡量算法的性能，上传失败时长。上传失败时长定义为丢帧数对应的播放时长，计算方式为丢帧数除以每秒的帧数。

三种算法中，默认的OBS丢帧算法丢帧数最多，上传失败时长最长，高达10秒，GreedyDrop智能丢帧算法减少了15%的上传失败，将上传失败时长降

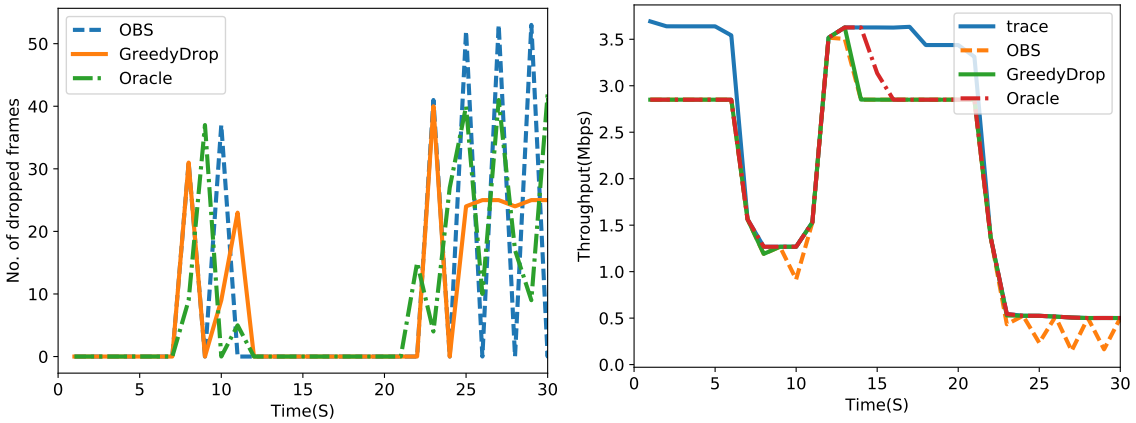
```
def traversal(item, index, count, front, Frame, buffer, remain, drop, dic, priority, send):
    if index<=count:#Frame backup
        Size = len(Frame)
        back = []
        for i in range(Size):
            if i!=0 and Frame[i]%GoP==0:#
                back = Frame[i:]
                break
            front.append(Frame[i])

        Size = len(front)
        for i in range(Size+1):
            New_front = front[:i]
            New_drop = drop + Size - i
            New_buffer = buffer - Size*Frame_Size + i*Frame_Size
            New_priority = priority
            if len(back)==0 and i!=Size:
                New_priority = True
            dic2 = traversal(item, index+1, count, New_front, back, New_buffer, remain, New_drop, dic, New_priority, send)
            dicMerged = dic.copy()
            dicMerged.update(dic2)
            dic = dicMerged.copy()
    else:
        Tuple = (tuple(front+Frame), buffer, remain, priority)
        #print Tuple
        From = item
        if dic.has_key(Tuple):
            From = (Record[Tuple] if dic[Tuple]<drop else item)
            send = (throughput[Tuple] if dic[Tuple]<drop else send)
            drop = min(dic[Tuple], drop)

        dic[Tuple] = drop
        Record[Tuple] = From
        throughput[Tuple] = send

    return dic
```

图 5.3　离线最优算法Oracle的具体实现



(a) 实时丢帧数　　(b) 实时的吞吐量

图 5.4　不同丢帧策略性能比较

低到9.13秒，改善丢帧的效果较为明显。而且GreedyDrop在线算法和离线最优Oracle算法之间的差距较小，约等于5%左右，离线最优算法的上传失败时长仅为8.8秒。实时丢帧数和真实吞吐量的时序图展示在图 5.4中。丢帧主要发生的时间段位于7到12秒和20到30秒期间。7到12秒时三种算法的表现基本相同，实时带宽紧跟控制数据一起变化；因为Oracle算法知道网络恢复的具体时间，相对于其他算法，会在视频队列中保存更多的有效帧，当网络状况恢复后，10到15秒期间时Oracle算法会将队列中的视频帧以burst的形式发送出去，相比于其他两个算

法，避免了一部分丢帧。20到30秒期间，网络经历着长时间的带宽抖动，OBS算法的丢帧数呈现周期性的振荡变化，方差很大，而GreedyDrop算法基本每个时刻的丢帧都很稳定。因为GreedyDrop算法只丢弃视频队列中属于旧的GoP的帧，而OBS算法每次丢帧将队列中的帧全部丢弃，因为丢帧行为会呈现一定的周期性。对于GreedyDrop算法，每个GoP开头的帧被发送出去，剩余的帧都被丢弃，规律性的行为使得丢帧数和吞吐量都保持恒定。Oracle算法和OBS算法的性能类似，每个时刻的丢帧数有一定的波动，但是方差变化比较小。同时考虑时间复杂度和算法性能，GreedyDrop是个不错的选择。

5.3　自适应码率算法

5.3.1　自适应码率算法实现

我们选取了几个在视频点播方面性能很好的动态码率算法，作为对比算法，来和GVBR算法做比较。为了控制变量，保持对比算法的一致性，和GVBR算法一样，对比算法中我们依然用调和平均值来作为对带宽的预估。

- OBS-VBR：简单直接的码率自适应算法，和OBS默认算法基本一致，使用OBS默认的丢帧策略，唯一的不同点在于动态码率，每个时刻都选择比估计带宽低的最高可用码率。
- MPC：用队列状态信息（帧数和帧的大小，以及帧的类型）和预测带宽值去计算接下来几个时隙的最优码率选择，但只应用第一个时隙的码率选择，对于之后的时隙重复上述过程。
- Robust-MPC：使用和MPC类似的方法，唯一的不同点，在于用过去几个时隙的最大预测误差去矫正未来几个时隙的带宽估计。

除了OBS-VBR算法之外，上述的几个对比算法均用GreedyDrop作为丢帧策略。Robust-MPC是视频点播领域最先进的码率自适应算法，MPC算法和Robust-MPC算法均是使用模型预测控制理论去选择未来时隙的码率。模型预测控制理论，简称MPC，主要的思想是预测未来几个时隙的带宽和系统状态，计算几个时隙的最优选择，但在应用时只用第一个时隙的选择，这个理论可以直接应用到直播场景中去，因此我们选择将Robust-MPC和MPC作为我们的直播动态码率对比算法。

GVBR算法具体实现的流程图如图 5.5所示。每当新的视频帧产生时，触发图中的机制去处理视频帧。本文中码率选择的粒度为一个GoP，所以如果新产生的视频帧是关键帧I帧的话，那么必须进行的过程是为该GoP选择一个合适的码率。码率的选择需要根据历史带宽信息以及视频帧剩余的数据量大小来确定。记录过

去5个GoP期间的真实带宽$[C_{j-5}, C_{j-1}]$，求其调和平均值C_j，作为该时隙的预估带宽。C_j的具体计算公式如下：

$$C_j = 1/(\sum_{k=j-5}^{k=j-1} 1/C_k) \tag{5-1}$$

。另外，考虑到视频帧队列里剩余的数据量$Rest$，循环扫描整个比特率数组，找到一个最大的码率$Rate$使得满足以下约束条件：

$$0.9 * Rate + Rest < C_j \tag{5-2}$$

。当选定当前GoP的码率值后，剩下的操作和视频帧为非关键帧相同。首先，在满足带宽约束的条件下，尽可能多的发送队列里的视频帧；之后，判断目前的视频帧队列是否满足及时性要求，即队列里是否存在$T1$时间之前产生的视频帧，弱存在，丢弃所有$T1$之间的视频帧。同时为了保证发送的视频帧都具有可解码性，丢弃和前一步丢帧有约束关系的视频帧。上述过程完整的叙述了视频帧产生后的处理机制，处理完成后，GVBR算法等待下一个视频帧的产生。

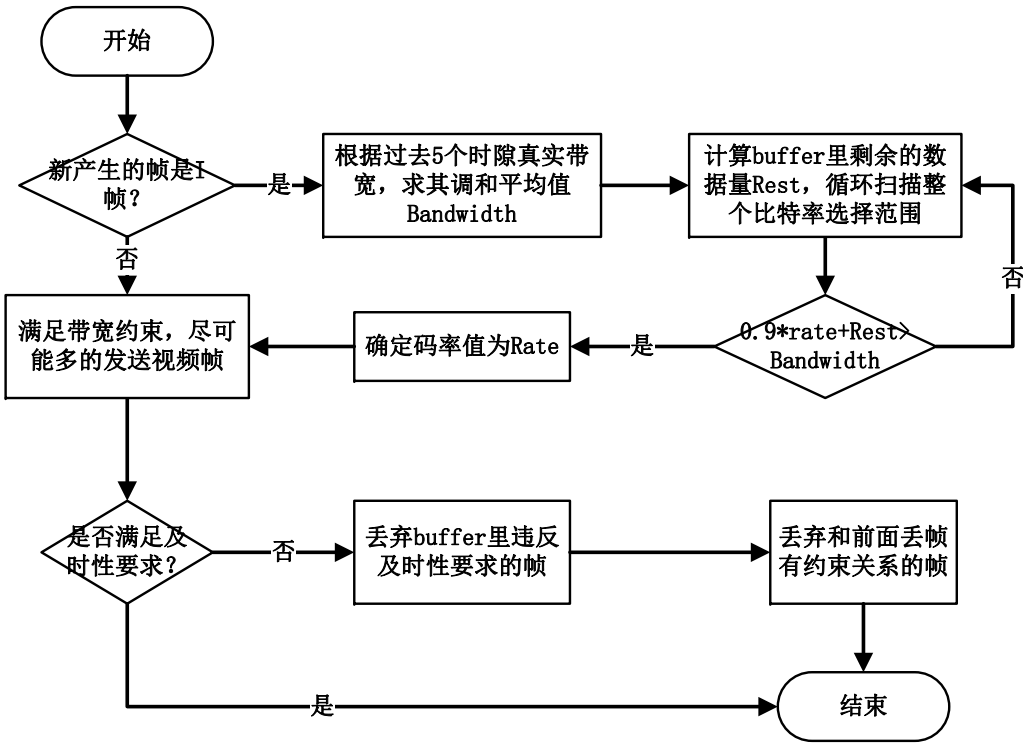


图 5.5　GVBR算法的实现流程图

Robust-MPC算法相比于MPC算法增加了带宽纠正的部分，用过去几个时隙的

预测误差去纠正当前时隙MPC算法对于带宽的预估，具体的代码实现如图 5.6。$Esm_history$为记录的过去τ个时隙的真实带宽值，$Error$为对于预测带宽的纠正参数。$Error$等于过去τ个时隙中最大的预测误差比，如代码所示。由于增加了对于带宽预估的纠正，Robust-MPC相对于MPC算法来说，性能得到明显的改善。

```
predict_Bandwidth = predict_bandwidth(Bw_history, Error)
for j in range(GoP):
    Esm_history.append(predict_Bandwidth[0])
if index >= tau:
    Bw_history = Bandwidth[(index+1-tau)*GoP:(index+1)*GoP]
    for j in range(GoP):
        Esm_history.remove(Esm_history[0])
else:
    Bw_history = Bandwidth[0:(index+1)*GoP]

Error = 0.0
for j in range(len(Esm_history)):
    if j%GoP==0:
        Error = max(Error, abs(Esm_history[j]-Bw_history[j])/Bw_history[j])
```

图 5.6　Robust-MPC算法中带宽纠正的代码实现

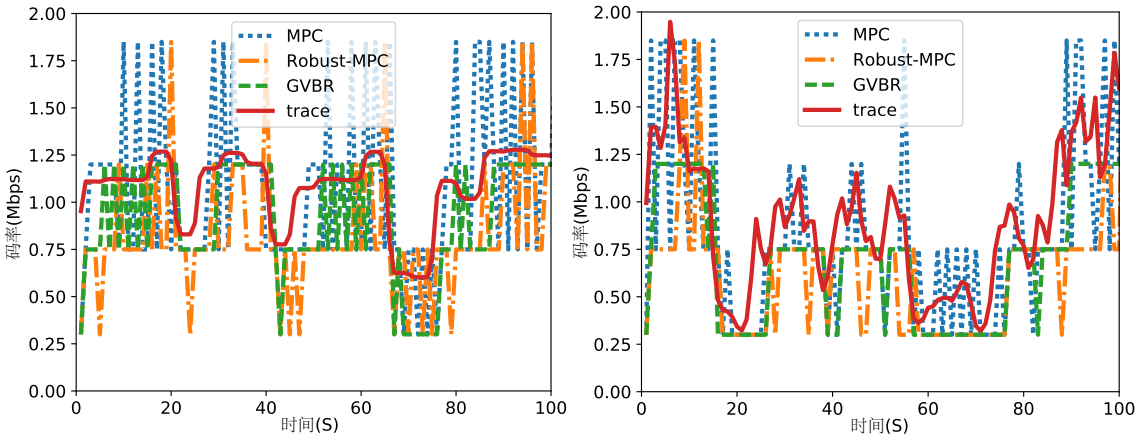
图 5.7是MPC算法和Robust-MPC算法的公共部分，给出了在未来τ个时隙如何选择最优的码率组合。对于未来的τ个时隙，假设每个时隙可选择的码率有M个，所有的码率组合数达到M^{τ}。循环扫描所有的码率组合，对于任一码率组合，模拟未来τ个时隙的具体过程，记录总的丢帧数，计算每一码率组合可以获得的总效益。找出最大效益对应的码率组合，并在接下来的一个GoP应用该码率选择，循环上述过程。MPC和Robust-MPC的具体实现就是这样。

5.3.2　码率自适应算法效果评估

详细的对比算法结果展示在图 5.8中。图 5.8(a)和图 5.8(b)分别展示了FCC数据集和HSDPA数据集下三种算法的性能。在这两幅图中，实线代表着真实世界的带宽数据记录。相比于Robust-MPC，MPC缺少对预测误差的矫正，容易更激进的选择更高的码率，反映到图中，MPC每个时刻的码率总是高于Robust-MPC的码率。除此之外，两个MPC算法的码率总是在真实带宽附近波动。两种数据集下，MPC和Robust-MPC的码率切换都比GVBR更频繁。这是因为GVBR的策略是选择比带宽低的最高码率，当带宽波动较小时，GVBR很有发生变化。但是MPC的策略是选择一个使用户体验质量最高的码率，当带宽稍微波动一点，MPC会更加倾向于选择一个更高或码率，增大优化目标中的第一项，码率相关的效益。而Robust-MPC虽然增加了对于预测误差的矫正，但依然是求几个时隙的最优选择，和MPC的行为较为类似。

```
def mpc_cal(Bandwidth, total, rate, buffer, remain, Send, Size, index, M, final_rate):
    if total==tau:
        for i in range(tau): #choose the tau-th bitrate
            priority = False
            Frame = rate[i]//fps
            Num = Num + rate[i]/10000.0/tau
            for j in range(GoP):
                if priority: ### drop unencodable frames
                    drop = drop + 1
                else:
                if remain<Bandwidth[j+i*GoP]:
                    rest = Bandwidth[j+i*GoP] - remain ### send frames as many as possible
                    while len(Send) and rest>0:
                    remain = 0
                else:
                    remain = remain - Bandwidth[j+i*GoP]
                buffer = max(buffer-Bandwidth[j+i*GoP],0)
                if len(Send) and GoP*(index+i)+j- min(Send)+1>=Bmax: ## voliate the timeliness rule, drop strategy
                count=0
                if priority: ## drop unencodable frames
                    for block in range(len(Send)):
                Send = Send[count:]
                Size = Size[count:]
        Num = Num - beta*drop
        return (final_rate,M)
    else:
        for i in range(len(Bitrate)): ### search all the possible choice tau times
            final_rate,M = mpc_cal(Bandwidth,total+1,rate,buffer,remain,New_Send,New_Size,index,M,final_rate)
        return (final_rate,M)
```

图 5.7　MPC和Robust-MPC未来几个时隙最优码率选择的代码实现



(a) FCC数据集的实时吞吐量　　(b) HSDPA数据集的实时吞吐量

图 5.8　不同码率自适应算法吞吐量比较

大规模的仿真实验的具体结果展示在图 5.9中，我们结合两个数据集去综合评估GVBR算法的性能，两个数据集分别是FCC数据集和HSDPA数据集。图中展示的三项指标，平均码率，上传失败时长以及用户体验质量都是归一化之后的平均指标。综合对比四种算法，MPC算法拥有最高的码率效益，这是因为MPC的带宽估计算法很激进，会更倾向于选择较高的码率。其他三种算法有着基本相同的平均码率，其中GVBR和Robust-MPC两种算法的平均码率稍微高一点，高于OBS-VBR算法。虽然MPC达到了最高的码率水平，但也带来了最高的丢帧数，对应的平均上传失败的时间也最长。OBS-VBR算法使用的丢帧算法是默认的OBS丢帧策略，所以在剩余三种算法中上传失败的时长最高。Robust-MPC算法相对于MPC算

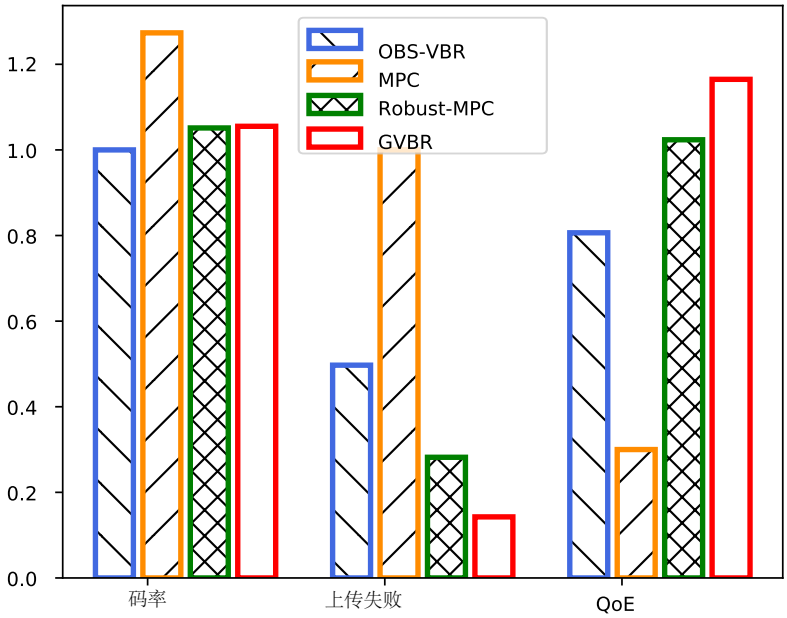
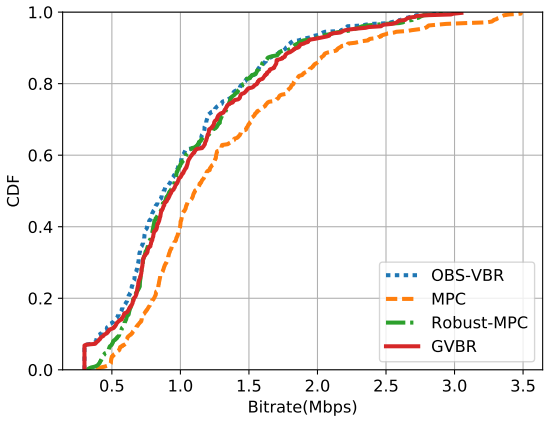


图 5.9　归一化平均码率，上传失败时长和用户体验质量

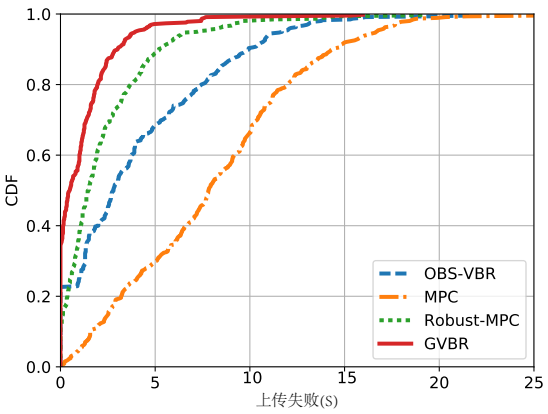
法来说增加了对于预测误差的纠正，将上传失败时间减少到可以忍受的时长。因为考虑了队列中排队等待发送的视频帧的数据量，GVBR将平均上传失败的时长减少到很少的时间，相对于Robust-MPC减少了50%左右。平均码率略微高于多个对比算法，而且平均上传失败时长很短，GVBR的用户体验质量在所有算法中达到了最高水平。剩余三种算法的用户体验质量排序分别是Robust-MPC，OBS-VBR和MPC。

平均码率，平均上传失败时长和平均用户体验质量的累计分布图展示在图5.10中。在图 5.10(a)中，GVBR算法位于Robust-MPC和OBS算法的右边，码率值稍微高于这两者。 MPC位于其余三种算法的右边，码率值远远高于其他三种算法，和图 5.9的结论一致。从图 5.10(b)中可以看出，对于GVBR算法，98%的实验其上传失败时间都低于5秒，有大约40%的的实验中未出现过上传失败。而其余三种算法低于5秒上传失败时间的百分比分别是，Robust-MPC有90%的实验上传失败时长低于5秒，OBS-VBR有70%的实验，MPC只有30%的实验。图 5.10(c)显示，GVBR算法只有2%的实验中用户体验质量不是很好，其余的98%的情况下归一化用户体验质量大约在$[0.8, 1]$之间。剩余三种算法的归一化用户体验质量大于80%的比例分别是，MPC为30%，OBS-VBR为70%，Robust-MPC为90%。总而言之，我们的GVBR算法在上传失败时长和用户体验质量两个指标上远远优于其他几个算法。

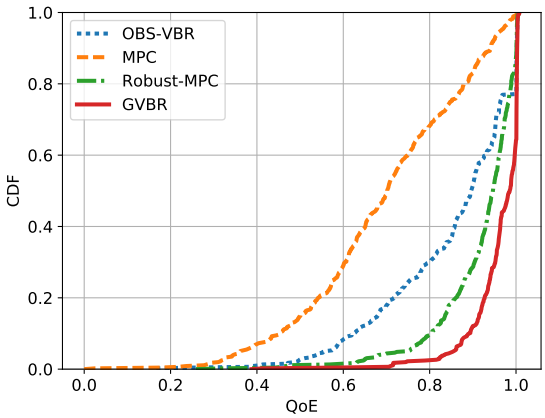
由于固定码率的OBS默认算法和其他四种自适应码率算法的性能差别过大，上述性能比较图中并未出现与默认OBS算法的对比结果。我们将默认OBS算法和



(a) 平均码率累计分布图



(b) 上传失败时长累计分布图



(c) 归一化用户体验质量累计分布图

图 5.10　大规模实验结果图

表 5.2　码率自适应算法效果对比

算法	平均码率(Mbps)	平均上传失败时长（秒）	平均用户体验质量
OBS	1.0788	26.1438	-39107.835
OBS-VBR	1.0256	3.9763	-5862.833
MPC	1.3060	8.0017	-11877.126
Robust-MPC	1.0781	2.2553	-3281.921
GVBR	1.0821	1.1414	-1605.033

几种动态码率算法的对比结果整理在表 5.2中。相比于最原始的固定码率的OBS算法，我们提出的GVBR算法减少了95.6%的丢帧数。默认的OBS算法其平均上传失败时长为26秒左右，而我们的GVBR算法只有1秒的平均失败时长。所有的算法除了MPC算法之外，GVBR算法在其中拥有最高的平均码率，略微高于其他三种算法。总的来说，我们的GVBR算法在达到更高的码率的基础上，显著减少了上传失败现象的发生。另外，表 5.2中可以得到的重要结论是，在主播端引入自适应码

率算法之后可以大幅度减少上传失败现象的发生。

5.4　本章小结

本章对于提出的优化方案进行了系统的性能评估，主要包括不同层面的方案，GoP层面的减少关键帧间隔以及码率自适应算法，GoP内部的智能丢帧策略。

首先，本章针对关键帧间隔大小对视频质量的影响进行了评估，发现了当关键帧间隔减少到一定值之后，视频质量呈现急剧下降，严重损害用户的观看体验。为了避免这种现象，关键帧间隔在选取时应严格遵循最小取值的约束，减少丢帧现象发生的同时保证视频质量。

其次，对于棘手的丢帧策略选择，我们对比了离线最优算法和GreedyDrop算法的优劣。相比于默认的OBS丢帧策略，GreedyDrop和离线最优算法的性能明显优于前面两者。但是离线最优算法的时间复杂度为指数级，对于移动设备的计算性能要求太高，而且离线最优算法需要已知全程的带宽状况，这在实际中是不可行的。综合时间复杂度和对于减少丢帧的性能改善，GreedyDrop算法是较优的选择。

最后，对于我们最后提出的一整套的解决方案GVBR，进行了充分的性能评估。不仅与视频点播领域较为先进的码率自适应算法MPC对比，而且和最原始的默认OBS算法进行了对比。实验结果显示，除了在平均码率的指标上略逊色于MPC算法，在平均上传失败时长和用户体验质量方面远远优于其他几种算法。

第6章　总结和展望

6.1　论文工作总结

随着近几年移动设备的普及和通信技术的发展，移动网络直播的用户规模呈现爆炸式增长，研究如何优化移动直播的用户体验质量越来越重要。移动直播中主播端因为占据着源端的至关重要地位，主播端在上传过程中发生的任何延迟和失败都会对所有的观看用户产生影响。但是移动主播一般都处在无线网络环境下，网络状况较为较为复杂。无线网络环境下移动主播经常需要和其他人一起竞争带宽，同时由于直播过程中主播的移动，周围环境的信号干扰等各种原因，带宽抖动的情况经常会发生。而且移动直播要求用户和主播间有高互动性，端到端的时延最多为几秒，这些因素都为优化用户体验带来了很大的挑战。本文对移动网络环境下主播端存在的性能问题进行研究，提出了一整套的协同解决方案，旨在通过优化主播端的传输性能去优化用户端的体验质量。

首先，本文通过对搭建的demo直播系统的性能测量，发现了移动直播过程中由于网络带宽抖动会导致两种质量问题，分别是短时的带宽抖动带来的应用层丢帧放大效应，以及长时间的带宽抖动带来的上传视频质量降低等问题。为了验证商业直播平台是否也存在类似的问题，我们选取了2个推流端，2个视频服务器组合起来，测量无线网络环境下商业平台的性能。通过实际的测量发现，商业平台也存在以上的两种问题，并不能很好的解决无线网络带来的质量问题。

为了解决上述问题，我们从三个方向进行了探索，涉及了GoP内部优化和GoP整体优化。我们提出的GVBR算法，是一整套提高主播端用户体验质量的协同解决方案，它修改了关键帧间隔的取值，改进了默认的丢帧策略，并且设计了一个有效的码率自适应算法。关键帧间隔的取值综合考虑了视频质量和丢帧现象等两个方面的影响，最后权衡两者给出了关键帧间隔的参考范围。修改后的丢帧策略GreedyDrop考虑了两个或多个GoP同时存在于视频队列中的情况，针对这种情况，GreedyDrop只丢弃旧的GoP的视频帧，保留较新的GoP相关的帧。GreedyDrop算法与离线最优算法Oracle之间的差距很小，只有5%的差距，而且同时GreedyDrop时间复杂度也很小，为线性时间复杂度。GVBR则根据估计出的带宽和视频队列数据量的差值去选择码率，考虑到视频队列里的剩余数据量可以更精确的计算真实可用的带宽。实验结果表明GVBR相比于最先进的码率自适应算法减少了50%的丢帧。对比最原始的默认OBS算法，我们提出的一整套协同解决方案，GVBR，将原始的上传失败时长从26秒减少到1秒，改进了95.6%，同时大

幅度改善了用户的体验质量。

6.2　未来工作展望

本文对个人交互直播主播端的用户体验优化研究仍存在不少欠缺的地方，后续的工作可以从下面两个可能的方向来进一步开展：

使用新的带宽预测算法提升带宽预测的准确率。因为本文主要研究的重点是优化主播端的传输性能，所以只是使用了一个现有的带宽预测方案。如果能够进一步提高带宽预测的准确度，相关算法的性能都会更上一个台阶。因此研究一个更加精确的带宽预测方案很有必要，这也是我们下一步工作的重点。

将GVBR算法在实际系统中实现。GVBR算法在设计时就考虑了实际部署的问题，算法的时间复杂度并不高，在实际部署中不会给移动设备带来很大的性能开销。但实际系统运行的情况远比理论的情况要复杂得多，所以GVBR算法可能在未来应该更结合实际情况，去做一些相应的修改。

参考文献

[1] CNNIC. 第41次《中国互联网络发展状况统计报告》[EB/OL]. http://www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/201803/P020180305409870339136.pdf.

[2] Cisco. Cisco visual networking index: Forecast and methodology, 2016–2021[EB/OL]. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html#_Toc484813971.

[3] 斗鱼. 斗鱼直播平台[EB/OL]. https://www.douyu.com/.

[4] 熊猫直播. 熊猫tv[EB/OL]. https://www.panda.tv/.

[5] Facebook. Facebook live[EB/OL]. https://live.fb.com/.

[6] Periscope. Periscope[EB/OL]. https://www.pscp.tv/.

[7] Zhang X, Liu J, Li B, et al. Coolstreaming/donet: A data-driven overlay network for peer-to-peer live media streaming[C]//INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE: volume 3. [S.l.]: IEEE, 2005: 2102–2111.

[8] Mukerjee M K, Naylor D, Jiang J, et al. Practical, real-time centralized control for cdn-based live video delivery[C]//ACM SIGCOMM Computer Communication Review: volume 45. [S.l.]: ACM, 2015: 311–324.

[9] Liao X, Jin H, Liu Y, et al. Anysee: Peer-to-peer live streaming[C]//INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings. [S.l.]: Citeseer, 2006: 1–10.

[10] Hei X, Liu Y, Ross K W. Inferring network-wide quality in p2p live streaming systems[J]. IEEE journal on Selected Areas in Communications, 2007, 25(9).

[11] Magharei N, Rejaie R, Guo Y. Mesh or multiple-tree: A comparative study of live p2p streaming approaches[C]//INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE. [S.l.]: IEEE, 2007: 1424–1432.

[12] Yin H, Liu X, Zhan T, et al. Design and deployment of a hybrid cdn-p2p system for live video streaming: experiences with livesky[C]//Proceedings of the 17th ACM international conference on Multimedia. [S.l.]: ACM, 2009: 25–34.

[13] Zhang C, Liu J. On crowdsourced interactive live streaming: a twitch. tv-based measurement study[C]//Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video. [S.l.]: ACM, 2015: 55–60.

[14] Twitch. twitch.tv[EB/OL]. https://www.twitch.tv/.

[15] Siekkinen M, Masala E, Kämäräinen T. A first look at quality of mobile live streaming experience: the case of periscope[C]//Proceedings of the 2016 Internet Measurement Conference. [S.l.]: ACM, 2016: 477–483.

[16] Wang B, Zhang X, Wang G, et al. Anatomy of a personalized livestreaming system[C]//Proceedings of the 2016 Internet Measurement Conference. [S.l.]: ACM, 2016: 485–498.

[17] Meerkat. Meerkat(app)[EB/OL]. https://en.wikipedia.org/wiki/meerkat.

[18] Li B, Wang Z, Liu J, et al. Two decades of internet video streaming: A retrospective view[J]. ACM transactions on multimedia computing, communications, and applications (TOMM), 2013, 9(1s): 33.

[19] Apple. Apple http live streaming[EB/OL]. https://developer.apple.com/streaming/.

[20] Stockhammer T. Dynamic adaptive streaming over http–: standards and design principles[C]// Proceedings of the second annual ACM conference on Multimedia systems. [S.l.]: ACM, 2011: 133–144.

[21] Sodagar I. The mpeg-dash standard for multimedia streaming over the internet[J]. IEEE MultiMedia, 2011, 18(4): 62–67.

[22] Wei S, Swaminathan V. Low latency live video streaming over http 2.0[C]//Proceedings of Network and Operating System Support on Digital Audio and Video Workshop. [S.l.]: ACM, 2014: 37.

[23] Swaminathan V, Wei S. Low latency live video streaming using http chunked encoding[C]// Multimedia Signal Processing (MMSP), 2011 IEEE 13th International Workshop on. [S.l.]: IEEE, 2011: 1–6.

[24] Hannuksela M M, Wang Y K, Gabbouj M. Random access using isolated regions[C]//Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on: volume 3. [S.l.]: IEEE, 2003: III–841.

[25] Bouzakaria N, Concolato C, Le Feuvre J. Overhead and performance of low latency live streaming using mpeg-dash[C]//Information, Intelligence, Systems and Applications, IISA 2014, The 5th International Conference on. [S.l.]: IEEE, 2014: 92–97.

[26] Mao H, Netravali R, Alizadeh M. Neural adaptive video streaming with pensieve[C]//Proceedings of the Conference of the ACM Special Interest Group on Data Communication. [S.l.]: ACM, 2017: 197–210.

[27] Akhshabi S, Begen A C, Dovrolis C. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http[C]//Proceedings of the second annual ACM conference on Multimedia systems. [S.l.]: ACM, 2011: 157–168.

[28] Petrangeli S, Famaey J, Claeys M, et al. Qoe-driven rate adaptation heuristic for fair adaptive video streaming[J]. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 2016, 12(2): 28.

[29] Jiang J, Sekar V, Zhang H. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive[J]. IEEE/ACM Transactions on Networking (TON), 2014, 22(1): 326–340.

[30] Li Z, Zhu X, Gahm J, et al. Probe and adapt: Rate adaptation for http video streaming at scale [J]. IEEE Journal on Selected Areas in Communications, 2014, 32(4): 719–733.

[31] Akhshabi S, Anantakrishnan L, Begen A C, et al. What happens when http adaptive streaming players compete for bandwidth?[C]//Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video. [S.l.]: ACM, 2012: 9–14.

[32] Huang T Y, Johari R, McKeown N, et al. A buffer-based approach to rate adaptation: Evidence from a large video streaming service[J]. ACM SIGCOMM Computer Communication Review, 2015, 44(4): 187–198.

[33] Yin X, Jindal A, Sekar V, et al. A control-theoretic approach for dynamic adaptive video streaming over http[C]//ACM SIGCOMM Computer Communication Review: volume 45. [S.l.]: ACM, 2015: 325–338.

[34] Nihei K, Yoshida H, Kai N, et al. Qoe maximizing bitrate control for live video streaming on a mobile uplink[C]//Telecommunications (ConTEL), 2017 14th International Conference on. [S.l.]: IEEE, 2017: 91–98.

[35] Pires K, Simon G. Dash in twitch: Adaptive bitrate streaming in live game streaming platforms [C]//Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming. [S.l.]: ACM, 2014: 13–18.

[36] De Cicco L, Mascolo S, Palmisano V. Feedback control for adaptive live video streaming[C]// Proceedings of the second annual ACM conference on Multimedia systems. [S.l.]: ACM, 2011: 145–156.

[37] Yu E, Jung C, Kim H, et al. Impact of viewer engagement on gift-giving in live video streaming [J]. Telematics and Informatics, 2018.

[38] Krasic C, Walpole J, Feng W c. Quality-adaptive media streaming by priority drop[C]// Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video. [S.l.]: ACM, 2003: 112–121.

[39] Singh S K, Leong H W, Chakravarty S N. A dynamic-priority based approach to streaming video over cellular network[C]//Computer Communications and Networks, 2004. ICCCN 2004. Proceedings. 13th International Conference on. [S.l.]: IEEE, 2004: 281–286.

[40] Huang J, Krasic C, Walpole J. Adaptive live video streaming by priority drop[M]. [S.l.: s.n.], 2003.

[41] 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]//15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). [S.l.]: USENIX Association, 2018: 267–282.

[42] OBS. Obs project[EB/OL]. https://obsproject.com/.

[43] dummynet. dummynet[EB/OL]. http://info.iet.unipi.it/~luigi/dummynet/.

[44] FCC. Measuring fixed broadband report-2016[EB/OL]. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-report-2016#block-menu-block-4.

[45] Riiser H, Vigmostad P, Griwodz C, et al. Commute path bandwidth traces from 3g networks: analysis and applications[C]//Proceedings of the 4th ACM Multimedia Systems Conference. [S.l.]: ACM, 2013: 114–118.

[46] Hore A, Ziou D. Image quality metrics: Psnr vs. ssim[C]//Pattern recognition (icpr), 2010 20th international conference on. [S.l.]: IEEE, 2010: 2366–2369.

[47] Shea R, Liu J, Ngai E C H, et al. Cloud gaming: architecture and performance[J]. Ieee Network, 2013, 27(4): 16–21.

[48] x264. x264[EB/OL]. https://www.videolan.org/developers/x264.html.

[49] Lai Z, Hu Y C, Cui Y, et al. Furion: Engineering high-quality immersive virtual reality on today's mobile devices[C]//Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking. [S.l.]: ACM, 2017: 409–421.

[50] Xiph.org. Xiph.org video test media [derf's collection][EB/OL]. https://media.xiph.org/video/derf/.

致　谢

衷心感谢我的导师崔勇教授对我在求学期间的精心指导和细心培养。崔老师不仅教会了我们科研的一些方式方法，还一直劝诫我们，无论是在科研生活中，还是在将来的职场工作中，都应该格外注重情商的培养。从崔老师这里，我学会了如何更好的和人打交道，相信我在将来的生活中会受益匪浅。在这里我向导师表达深深的谢意。

另外，感谢交叉信息研究院的吴文斐老师对我的指导。每次找文斐老师讨论论文的相关事情，他总是很积极响应，在论文的很多方面都帮了我不少大忙。感谢芝加哥大学的江鋆晨老师对我的指导，很多次即使在机场，也和我讨论论文工作。

感谢实验室的同学们对我的关心和帮助。其中，尤其是宋健师兄。在和宋健师兄一起合作的两年里，他坚持不懈的科研精神深深的触动了我；另外，宋健师兄总是像一个长辈一样，帮我解答了许多生活上的问题。感谢亚运，扬军，莫为，楚鸣和范权，实验室的生活有你们不再枯燥。另外也感谢同级的杨振杰，戴柠薇，许哲和孙霖辉，感谢那段我们一起奋斗的时光。

感谢论文评审老师和答辩组的老师百忙之中依然评审我的论文，谢谢。

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、在学期间发表的学术论文与研究成果

个人简历

1992 年 8 月 16 日出生于安徽省淮北市。

2011 年 9 月考入北京邮电大学信息与通信工程学院系通信工程专业，2015 年 7 月本科毕业并获得工学学士学位。

2015 年 9 月免试进入清华大学计算机科学与技术系攻读硕士学位至今。

发表的学术论文

[1] Qingmei Ren, Yong Cui, Wenfei Wu, Changfeng Chen, Yuchi Chen, Jiangchuan Liu and Hongyi Huang. Improving Quality of Experience for Mobile Broadcasters in Personalized Live Video Streaming. Accepted by IEEE/ACM International Symposium on Quality of Service (IWQoS) 2018 Short Paper.

[2] Yong Cui, Jian Song, Kui Ren, Minming Li, Zongpeng Li, Qingmei Ren and Yangjun Zhang. Software defined cooperative offloading for mobile cloudlets[J]. IEEE/ACM Transactions on Networking (TON), 2017.

[3] Yong Cui, Jian Song, Minming Li, Qingmei Ren, Yangjun Zhang and Xuejun Cai. SDN-based big data caching in ISP networks[J]. IEEE Transactions on Big Data (TBD), 2017.

研究成果

[1] 崔勇, 宋健, 任青妹. 智能终端能耗优化的自适应流媒体分发方法: 中国, CN105245919B. (中国专利公告号.)