Crowdsourced Livecast Systems: Measurement and Enhancement

by

Cong Zhang

M.Sc., Zhengzhou University, 2012
B.Sc., Information Engineering University, 2008

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

© Cong Zhang 2018
SIMON FRASER UNIVERSITY
Spring 2018

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Approval

Name:	**Cong Zhang**
Degree:	**Doctor of Philosophy (Computing Science)**
Title:	**Crowdsourced Livecast Systems: Measurement and Enhancement**
Examining Committee:	**Chair:** Dr. Ryan Shea Assistant Professor
	Dr. Jiangchuan Liu Senior Supervisor Professor
	Dr. Qianping Gu Supervisor Professor
	Dr. Kangkang Yin Internal Examiner Associate Professor School of Computing Science
	Dr. Yonggang Wen External Examiner Associate Professor School of Computer Science and Engineering Nanyang Technological University
Date Defended:	**March 01, 2018**

Abstract

Empowered by today's rich tools for media generation and collaborative production, multimedia service paradigm is shifting from the conventional single source, to multi-source, to many sources, and now towards crowdsource, where the available media sources for the content of interest become highly diverse and scalable. Such crowdsourced livecast systems as Twitch.tv, YouTube Gaming, and Periscope enable a new generation of user-generated livecast systems, attracting an increasing number of viewers all over the world. Yet the sources are managed by unprofessional broadcasters, and often have limited computation capacities and dynamic network conditions. They can even join or leave at will, or crash at any time.

In this thesis, we first conduct a systematic study on the existing crowdsourced livecast systems. We outline the inside architecture using both the crawled data and the captured traffic data from local broadcasters/viewers. We then reveal that a significant portion of the unpopular and dynamic broadcasters are consuming considerable system resources. Because cloud computing provides resizable, reliable, and scalable bandwidth and computational resources, which naturally becomes an effective solution to leverage heterogeneous and dynamic workloads. Yet, it is a challenge to utilize the resources from the cloud cost-effectively. We thus propose a cloud-assisted design to smartly ingest the sources and cooperatively utilize the resources from dedicated servers and public clouds.

In current crowdsourced livecast systems, crowdsourced gamecasting is the most popular application, in which gamers lively broadcast game playthroughs to fellow viewers using their desktop, laptop, even mobile devices. These gamers' patterns, which instantly pilot the corresponding gamecastings and viewers' fixations, have not been explored by previous studies. Since mobile gamers and eSports gamers occupy a large portion of content generators. In this thesis, we target on two typical crowdsourced gamecasting scenarios, i.e., mobile gamecasting and eSports gamecasting, respectively. We investigate the gamers' patterns to explore their effects on viewers and employ intelligent approaches, e.g., learning-based techniques, to capture the associations between gamers' patterns and viewers' experiences. Then, we employ such associations to optimize the streaming transcoding and distribution.

Keywords: Crowdsourced livecast; measurement; enhancement

Dedication

To my family !

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my senior supervisor, Dr. Jiangchuan Liu for his constant support and guidance throughout my PhD studies. His warm encouragement, thoughtful advice, and life wisdom also motivate me to become a good "game player".

I am also grateful to Dr. Qianping Gu, Dr. Kangkang Yin, and Dr. Yonggang Wen for serving in my examining committee. I thank them for their precious time on reviewing my work and for their advices on improving my thesis. I would like to thank Dr. Ryan Shea for charing my PhD thesis defence.

I thank a number of colleagues and friends from their help and support during my stay at Simon Fraser University. In particular, I thank Dr. Feng Wang, Dr. Haiyang Wang, Dr. Fei Chen, Dr. Xiaoqiang Ma, Lei Zhang, Xiaoyi Fan, Qiyun He, Jia Zhao, Yifei (Stephen) Zhu and Silvery (Di) Fu, for helping me with the research as well as many other problems during my PhD study.

Last but not least, I wish to thank my family for their love and support. This thesis is dedicated to you all !

Table of Contents

Approval **ii**

Abstract **iii**

Dedication **iv**

Acknowledgements **v**

Table of Contents **vi**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Overview of Video Streaming 2

1.2 Overview of Crowdsourced Livecast 3

1.3 Contributions 5

1.4 Thesis Organization 6

2 A Twitch.TV-Based Measurement Study **8**

2.1 Inside the Twitch Architecture 9

2.2 View Statistics and Patterns 12

2.2.1 Popularity and Duration 12

2.2.2 Event- and Source-Driven Views 14

2.3 Messaging and View Latency 15

2.4 Live Messaging Latency 16

2.4.1 Broadcast Latency 16

2.4.2 Source Switching Latency 17

2.4.3 Impact of Broadcaster's sources 17

2.5 Summary 18

3 Cloud-assisted Crowdsourced Livecast **19**

3.1 Related Work 20

3.2 Measurements of Crowdsourced Livecast: Twitch as a Case Study 22
3.2.1 Twitch-based Datasets 22
3.2.2 Characteristics of Crowdsourced Live Broadcasters 24
3.2.3 Effects of Crowdsourced Live Events 24
3.2.4 Popularity of Crowdsourced Live Broadcasters 25
3.2.5 Dynamics of Crowdsourced Live Broadcasters 27
3.2.6 Challenges of Hosting Unpopular Broadcasters 28
3.3 CACL: Architecture and Design 28
3.3.1 EC2-based measurement 28
3.3.2 Round-trip Time 29
3.3.3 Broadcast Latency 30
3.3.4 CACL Architecture 31
3.3.5 Initial Offloading 32
3.4 Problem Formulation and Solution 32
3.4.1 Basic Model with Ingesting Latency 33
3.4.2 Enhanced Model with Transcoding Latency 34
3.4.3 Solution 35
3.5 Performance Evaluation 37
3.5.1 Efficiency of Resource Allocation 37
3.5.2 Trace-driven Simulation 38
3.6 Summary 40

4 Exploring Viewer Gazing Patterns for Touch-based Mobile Gamecasting 41
4.1 Background 42
4.2 Motivation 45
4.3 Interaction-Aware Design 47
4.3.1 Touch-assisted Prediction Module 48
4.3.2 Tile-based Optimization Module 48
4.4 Understanding Game Touch Interactions 49
4.4.1 Touch Data Collection 49
4.4.2 Interaction Classification 50
4.5 Insights into Viewers' Gazing Patterns 51
4.5.1 Gazing Data Collection 51
4.5.2 Gazing Classification 52
4.6 Touch-Gaze Association Learning 54
4.6.1 Preliminaries 54
4.6.2 Touch-Gaze Association 54
4.7 Tile-based Optimization 56
4.7.1 Problem Formulation 56

4.7.2 Solution . . . 57
4.8 Performance Evaluation . . . 59
4.8.1 Performance of Touch-assisted Prediction . . . 59
4.8.2 Trace-driven Simulation . . . 60
4.8.3 User Study . . . 63
4.9 Summary . . . 64

5 Highlight-Aware eSports Gamecasting with Strategy-Based Prediction 65
5.1 Background . . . 67
5.1.1 Optimization in Crowdsourced Gamecasting Services . . . 70
5.1.2 Learning-based QoE Improvement in Multimedia Systems . . . 70
5.2 Measurement and Observation . . . 71
5.2.1 Measurement Settings . . . 71
5.2.2 Measurement Results . . . 72
5.3 StreamingCursor: An Overview . . . 72
5.4 Framework Design and Solution . . . 75
5.4.1 Strategy-based Prediction . . . 75
5.4.2 Transcoding Task Assignment Optimization . . . 76
5.5 Performance Evaluation . . . 79
5.5.1 Strategy-based Prediction Results . . . 79
5.5.2 Highlight-aware Optimization Performance . . . 81
5.6 Summary . . . 82

6 Conclusion 83
6.1 Summary of the Thesis . . . 83
6.2 Future Directions . . . 84
6.2.1 Further Measurements on Crowdsourced Gamecasting . . . 84
6.2.2 Further Enhancement on the Broadcaster Side . . . 84
6.2.3 Further Works on Mobile Gamecasing . . . 85

Bibliography 86

List of Tables

Table 2.1 Twitch REST APIs used in our crawler 9
Table 2.2 Configuration of broadcasters' devices 12
Table 2.3 Configuration of viewers' devices 12

Table 4.1 Classification and definition of gamer's touch interactions 42
Table 4.2 Classification and definition of viewer's gazing patterns 43
Table 4.3 Statistics of touch data 47
Table 4.4 Encoding example 54

Table 5.1 Configuration of Amazon EC2 instances 71
Table 5.2 Statistics of a replay file 75
Table 5.3 Performance of the strategy-based highlight prediction 79

List of Figures

Figure 1.1 An illustration of two crowdsourced live streams 4

Figure 2.1 Device distribution of Twitch's live sources 10
Figure 2.2 Two broadcasters/game players measurement configuration 11
Figure 2.3 Live streams rank ordered by views 13
Figure 2.4 Distribution of live streaming duration 13
Figure 2.5 Views patterns in Twitch (From 2014OCT01 to 2014OCT07) 13
Figure 2.6 The characteristics at the viewer-side 15
Figure 2.7 Impact of broadcaster's network 18

Figure 3.1 A generic system diagram of crowdsourced livecast platforms 21
Figure 3.2 Distribution of three types of devices (Eastern Standard Time) . . 23
Figure 3.3 Broadcasters' inter-arrival time 23
Figure 3.4 Broadcasters' arrival rate 23
Figure 3.5 Characteristics of crowdsourced live events 25
Figure 3.6 Broadcasters rank ordered by popularity 26
Figure 3.7 Distribution of livecast duration 26
Figure 3.8 Broadcaster arrivals per five minutes 26
Figure 3.9 Two examples in the broadcaster dataset 27
Figure 3.10 Different types of broadcasters' resource consumptions 28
Figure 3.11 Diagram of the EC2-based measurement 29
Figure 3.12 RTT comparison 29
Figure 3.13 Broadcast latency and CPU usage on different instances 30
Figure 3.14 Framework of Cloud-assisted Crowdsourced Livecast (CACL) . . . 31
Figure 3.15 Impact of computational capacity 37
Figure 3.16 Impact of threshold H 39
Figure 3.17 Performance evaluation of proposed solutions 39

Figure 4.1 A generic architecture of the MGC platform 43
Figure 4.2 Motivations of our study 44
Figure 4.3 Interaction-aware optimization framework in MGC 46
Figure 4.4 Sample of touch screen events 46
Figure 4.5 Time intervals between consecutive touch interactions 49

Figure 4.6 Characteristics of touch regions 50
Figure 4.7 Example of gazing points classification 51
Figure 4.8 Eye tracking device in our testbed 51
Figure 4.9 An illustration of collecting a viewer's gazing data 51
Figure 4.10 Characteristics of gazing regions 52
Figure 4.11 CDF of p-value 53
Figure 4.12 Encoding tree 53
Figure 4.13 Eye gazing patterns vs. Prediction results 59
Figure 4.14 Normalized Scanpath Saliency 60
Figure 4.15 Area Under the Curve (Borji) 60
Figure 4.16 Area Under the Curve (Judd) 60
Figure 4.17 Energy measurement platform 61
Figure 4.18 Impact of different α and β settings 61
Figure 4.19 Comparison of our approach and DASH (SRD) 62
Figure 4.20 PSNR experiments in four gazing pattern sets 63
Figure 4.21 Satisfaction Score (error bars are 95% confidence intervals) 63

Figure 5.1 Impact of transcoding stages 66
Figure 5.2 An illustration of CILS paradigm 68
Figure 5.3 Illustrations of the playing flow in Dota2. 69
Figure 5.4 Measurement results for the transcoding tasks on different instances 73
Figure 5.5 Workflow of the framework StreamingCursor 74
Figure 5.6 Design of highlight prediction 74
Figure 5.7 Example of task assignment 74
Figure 5.8 Normalized transcoding expense under different number of broadcasters 80
Figure 5.9 Normalized transcoding latency under different number of broadcasters 80

Chapter 1

Introduction

Empowered by today's rich tools for media generation and collaborative production, multimedia service paradigm is shifting from the conventional single source, to multi-source, to many sources, and now towards *crowdsource* [63], where the available media sources for the content of interest become highly diverse and scalable. Such crowdsourced livecast systems as *Twitch.tv* (or Twitch for short), *YouTube Gaming*[1], and *Periscope*[2] enable a new generation of user-generated livecast systems, attracting an increasing number of viewers all over the world. Crowdsourced content creation is expected to usher in a new wave of innovations in how multimedia content is created and consumed, allowing content creators from different backgrounds, talents, and skills to collaborate on producing future multimedia content.

Different from user-generated Video-on-Demand (VoD) services [13, 85, 87], e.g, YouTube and Vimeo, and professional broadcasting services, e.g., NBC (National Broadcasting Company) and CBC (Canadian Broadcasting Corporation), crowdsourced livecast systems do not provide the sources of live contents by themselves. Rather, they serve as the platforms that bridge the sources and viewers, thereby greatly expanding the content and user bases. On the other hand, the sources are managed by unprofessional broadcasters, and often have heterogeneous devices and networking conditions, which generate a large number of contents with different bit-rates and resolutions. They can even join or leave at will, or crash at any time. According to our measurement from Twitch, the number of the concurrent broadcasters is about 6 thousand at the quiet time and more than 25 thousand at the prime time. If these broadcasters create live contents simultaneously from their personal devices, this crowdsourced paradigm will lead to more dynamic workloads both in the ingesting and transcoding steps. All these make high-quality streaming more challenging.

There have been some studies on crowdsourced livecast systems [38, 59, 2, 6, 10, 72]. Many of them still consider these systems as the live streaming systems with a large-scale of viewers, but neglect the challenges from the heterogeneous broadcasters [38, 2]. The latest

[1] `https://gaming.youtube.com/`

[2] `https://www.pscp.tv/`

studies mainly focus on the following three directions: (1) how to minimize the latency gap between different viewers[10, 26]; (2) how to optimize the ingesting and transcoding services from the perspective of the crowdsourced livecast service providers [10, 27, 81, 78]; (3) how to reduce the encoding/uploading latency on the broadcaster-side [72, 86].

In this chapter, we first present the overview of video streaming and crowdsourced live streaming. Then, we summarize the contributions and describe the organization of this thesis.

1.1 Overview of Video Streaming

Due to the rapid growth of high-performance personal devices (e.g., smartphone) and the widespread deployment of high-speed communication networks (e.g., 4G/LTE), video streaming has become one of the most popular Internet services and attracted an increasing number of users as the content providers and consumers [54, 1, 33, 46, 16]. To serve the growing workloads, the architecture of video streaming services mainly experiences three stages [43]:

- Client-Server model: This model is a typical distributed structure used by a large number of applications, such as Email, World Wide Web (WWW), and FTP (File Transfer Protocol) downloading applications. In client-server video streaming applications, the original contents are managed and generated by professional broadcasting companys, such as BBC (British Broadcasting Corporation), these contents are distributed via Internet. During the 1990s and early 2000s, most of the researchers studied the design and implementation of new streaming protocols, such as Real Time Streaming Protocol (RTSP) [61], RTP Control Protocol (RTCP) [22], and Real-time Transport Protocol (RTP) [34]. The drawback of client-server video streaming applications is that the server node must maintain the streaming status of all requests from the client nodes, which consumes a huge number of resources when the client nodes increase largely.

- Peer-to-Peer (P2P) network: P2P network was popularized by file sharing applications in 1999. Then, it was used in video streaming applications to scale up an increasing number of users, attracting much attention from academia [83, 28, 49]. In a P2P video streaming application, each video client is considered as a peer, which not only downloads video contents from original servers, but also uploads these contents to other peers. The main issue of using P2P streaming protocol is that all peers need to install dedicated applications and this protocol is not firewall-friendly.

- HTTP video streaming: To conveniently transfer video contents via the Internet and users' local networks, HTTP video streaming was proposed and implemented recently. According to the latest HTTP video streaming standard DASH (Dynamic Adaptive

Streaming over HTTP [64, 67]), a video is encoded into various versions with different bit-rates and resolutions; each version is divided into a sequence of small segments. These segments can be downloaded using HTTP and distributed by standard Content Delivery Networks (CDNs). Video clients can adaptively adjust the streaming rates according to networking capacity, buffer size, etc [53, 84, 51].

Today, such commercial services as Netflix, YouTube, and Hulu have adopted HTTP video streaming to delivery their videos to the users. A substantial amount of researches focus on the optimization of streaming systems [4], the improvements of Quality-of-Service (QoS) [77, 48] and Quality-of-Experience (QoE) [21, 36] from service providers' and viewers' perspectives, respectively. Besides, social network services (SNS) [44] [50] and cloud-based technologies [7] [23, 31] provide lots of unique opportunities to improve the delivery of video streaming.

1.2 Overview of Crowdsourced Livecast

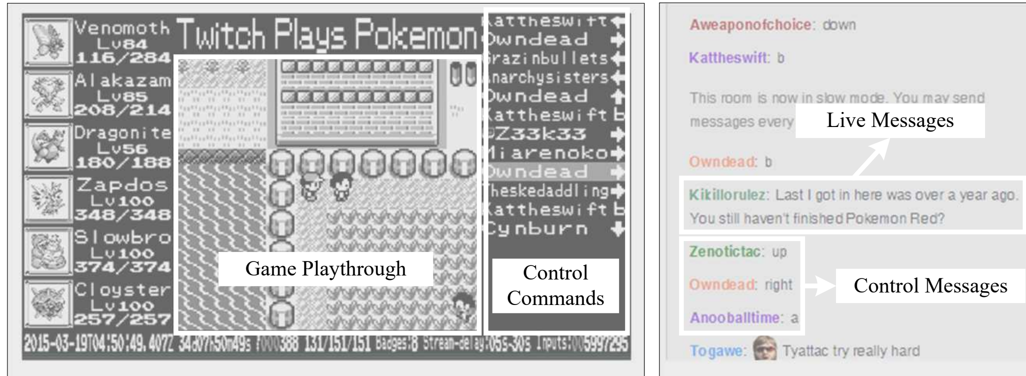
In this section, we briefly introduce the system diagram of crowdsourced livecast systems. As shown in Figure 3.1, two main services, *streaming service* and *chatting service*, jointly serve the geo-distributed broadcasters and fellow viewers. In the former, broadcasters' devices (i.e., sources) send encoded streams to the service provider's ingesting servers, using TCP-based protocols, e.g., Real Time Messaging Protocol (RTMP)[3], to maintain the low-latency communication. Then, the streams are transcoded to multi-quality formats, e.g., HTTP Live Streaming (HLS)[4], and delivered to fellow viewers through Content Delivery Networks (CDNs). In the latter, a set of chatting servers receive the viewer's live messages, and then dispatch the messages to the corresponding broadcaster and other viewers, enhancing the participants' experience and interaction in live events [80].

For example, the crowdsourced gamecasting "TwitchPlaysPokemon"[5], as shown in Figure 1.1a, offered the live stream and emulator for a role-playing video game "Pokemon Red", in which players (also as the viewers in Twitch) simultaneously send the control messages of Pokemon through the IRC (Internet Relay Chat) protocol and live messages in Twitch. That said, the viewers are no longer passive, but can affect the progress of the broadcast as well. This truly crowdsourced game streaming attracted more than 1.6 million players and 55 million viewers. Figure 1.1b demonstrates another typical livecast example, in which two travelers (i.e., broadcasters) are lively broadcasting their journey, performing their discussion and guitar show. We split this screenshot into three areas. The travelers'

[3] `https://en.wikipedia.org/wiki/Real-Time_Messaging_Protocol`

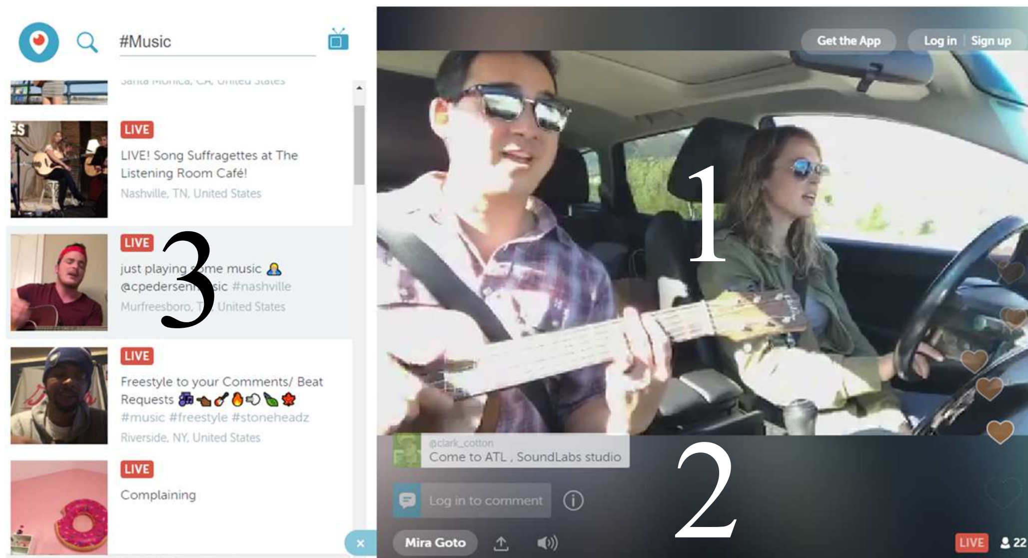
[4] `https://en.wikipedia.org/wiki/HTTP_Live_Streaming`

[5] `https://en.wikipedia.org/wiki/Twitch_Plays_Pokemon`



Streaming service　　　　Chatting service

(a) TwitchPlaysPokemon (Twitch)



(b) Daily Show (Periscope)

Figure 1.1: An illustration of two crowdsourced live streams

shows are rendered in area #1. The viewers can discuss this performance using the chatting window in area #2. In such a scenario, it is worth noting that the broadcasters and viewers can be highly heterogeneous and dynamic, who can have quite different hardware and software configurations, and may join or leave the system at will. The broadcasters' popularity varies significantly as well. "TwitchPlaysPokemon" has attracted more than 72 million viewers[6]; yet many broadcasters have only one or two viewers, or even none.

[6] `https://www.twitch.tv/twitchplayspokemon`

1.3 Contributions

In this thesis, we present a comprehensive study on crowdsourced livecast systems from the perspectives of measurement and enhancement. In particular, we make the following contributions:

- We for the first time present an initial investigation on crowdsourced livecast systems. Taking Twitch as a representative, we outline their inside architecture using both the crawled data and the captured traffic data from local broadcasters/viewers. After closely examining the access data collected in a two-month period, our measurement results reveal the unique source- and event-driven viewing features, showing that the existing delay strategy on the viewer's side substantially impacts the viewers' interactive experiences, and there is a significant disparity between the long broadcasting latency and the short live messaging latency. On the broadcaster's side, the dynamic uploading capacity is a critical challenge, which noticeably affects the smoothness of live streaming for viewers.

- We further analyze the popularity of different broadcasters and calculate their bandwidth and computational consumptions. The results show that a significant portion of the unpopular and dynamic broadcasters are consuming considerable system resources. Yet expensive server clusters have been deployed to ingest and transcode live streams, fulfilling the demands from a large number of heterogeneous broadcasters and geo-distributed viewers. Through the real-world measurement and data analysis, we show that the public cloud has great potentials to address these scalability challenges. We accordingly present the design of Cloud-assisted Crowdsourced Livecast (CACL) and propose a comprehensive set of solutions for broadcaster partitioning. Our trace-driven evaluations show that our CACL design can smartly assign ingesting and transcoding tasks to the elastic cloud virtual machines, providing flexible and cost-effective system deployment.

- We have witnessed an explosion of gamecasting applications, in which game players (or *gamers* in short) broadcast game playthroughs by their personal devices in real-time. Such pioneer platforms as YouTube Gaming, Twitch, and Mobcrush have attracted a massive number of online broadcasters, and each of them can have hundreds or thousands of fellow viewers. The growing number, however, has created significant challenges to the network and end-devices, particularly considering that bandwidth- and battery-limited smartphones or tablets are becoming dominating for both gamers and viewers. Yet the unique touch operations of mobile interface offer opportunities, too. In this paper, our measurements based on the real traces from gamers and viewers reveal that strong associations exist between the gamers' touch interactions and the viewers' gazing patterns. Motivated by this, we present a novel interaction-aware

optimization framework to improve the energy utilization and stream quality for mobile gamecasting. Our framework incorporates a touch-assisted prediction module to extract association rules for gazing pattern prediction and a tile-based optimization module to utilize energy on mobile devices efficiently. Trace-driven simulations illustrate the effectiveness of our framework in terms of energy consumption and stream quality. Our user study experiments also demonstrate much improved quality satisfaction over the state-of-the-art solution with similar network resources.

- As the most complicated and popular branch in crowdsourced livecast systems, eSports gamecasting service has attracted much attention from a large number gamers and viewers. To mitigate the huge pressure from gamecasting delivery, CGC service providers have to transcode gamers' RTMP (Real Time Message Protocol) streaming to HTTP-based live streaming. Yet more than 68% of the gamecasting latency between gamers and viewers come from the transcoding step according to the statistics from Twitch.tv. In this paper, we are interested in optimizing transcoding task assignment in eSport gamecasting services. To explore the challenges therein, we deploy a gamecasting testbed and find that game events in eSports games largely impact the complexity of game scenes, which in turn determines the transcoding latency of the corresponding gamecasting contents. Motivated by this observation, we propose a novel framework StreamingCursor, which first analyzes gamers' interactions and strategies to capture key game events (i.e., highlights) with the assistance of deep learning techniques, and then optimizes transcoding task assignment in eSports gamecasting service. Our design has been extensively evaluated through our trace-driven experiments.

1.4 Thesis Organization

The remainder of the thesis is structured as follows:

- In Chapter 2, we examine the characteristics of broadcasters and viewers based on the crawled data from Twitch.tv. In addition, we study the impact of heterogeneous devices on the broadcaster side and viewer side, respectively.
- In Chapter 3, based on our measurement study on different broadcasters, we propose a cloud-assisted design to allocate resources cost-effectively.
- In Chapter 4, we investigate the associations between the broadcasters and the viewers in mobile gamecasting applications. We further propose an interaction-aware design to predict such associations and employ them to optimize the tile-based streaming transmission.

- In Chapter 5, we analyze the relationships between different events and transcoding latencies in eSports gamecasting. Through investigating the potentials from the players' interactions, we build a learning-based framework to predict the game highlights and use the prediction results to assign the transcoding tasks efficiently.
- In Chapter 6, we conclude this thesis, and also discuss some future directions.

Chapter 2

A Twitch.TV-Based Measurement Study

Recent years, crowdsourced content creation is expected to usher in a new wave of innovations in how multimedia content is created and consumed, allowing content creators from different backgrounds, talents, and skills to collaborate on producing future multimedia content. For instance, the industrial pioneer, Twitch.tv (www.twitch.tv), allows anyone to broadcast their content to massive viewers, and the primary sources come from game players from PCs or other gaming consoles, e.g., PS4 and XBox. According to Twitch's Retrospective Report 2013[1], in just three years, the number of viewers grew from 20 million to 45 million, while the number of unique broadcasters tripled to 900 thousand. Other similar platforms such as Poptent (www.poptent.com) and VeedMe (www.veed.me) have emerged in the market with great success, too. In the 2014 Sochi Winter Olympics, NBC (National Broadcasting Company) had a total of 41 live feeds distributed both in Sochi and in USA, and in the 2014 FIFA World Cup, when a goal is scored, CBC (Canadian Broadcasting Corporation) synchronized the live scenes of the cheering fans in public squares from cities worldwide in its live streaming channel. The evolution is driven further by the advances in personal and mobile devices that can readily capture high-quality audio/video anywhere and anytime (e.g., iPhone 6 supports 60 fps 1080p High Definition (HD) video recording, and 240 fps slow-motion recording for 720p HD videos).

Crowdsourced livecast systems promote viewers' involvement with live content broadcasters. The viewers can choose their preferred perspective for a live event (e.g., one particular game player, or a game commentator) and enjoy virtual face-to-face interactions with real-time chatting. It is necessary to ensure timely interaction and minimize the switching latencies, which again is aggravated with the multiple non-professional sources and the massive viewers.

[1] http://www.twitch.tv/year/2013

In this chapter, we present an initial investigation on the modern crowdsourced livecast systems. Taking Twitch as a representative, we outline their inside architecture using both crawled data and captured traffic of local broadcasters/viewers. Closely examining the access data collected in a two-month period, we reveal that the view patterns are determined by both events and broadcasters' sources. Our measurements explore the unique source-driven and event-driven views, showing that the current delay strategy on the viewer's side substantially impacts the viewers' interactive experience, and there is significant disparity among the long broadcast latency and the short live messaging latency. On the broadcaster's side, the dynamic uploading capacity is a critical challenge, which noticeably affects the smoothness of live streaming for viewers. Inspired by the measurement results, we discuss potential enhancements toward better crowdsourced interactive live streaming.

2.1 Inside the Twitch Architecture

Table 2.1: Twitch REST APIs used in our crawler

REST APIs	Description
GET /streams/summary	Get the global statistics of streams and views at present
GET /streams	Get the meta file of live streams at present
GET /channels/:channel	Get the number of total views, followers and delay setting of broadcaster's channel
GET /channels/:channel/videos	Get the number of total views, duration of each stream in broadcaster's channel

As a new generation and proprietary system, despite certain information leakages [30], the inside details of Twitch and particularly the access data remain unclear to the public, so do many other crowdsourced livecast systems in the market. With the assistance of Twitch's Representational State Transfer (REST) APIs[2], we continually crawled the access data of live contents from Twitch in a two-month period (from October 1st to November 30th, 2014). The crawled data include the number of Twitch streams, the number of Twitch views, and the meta-data of live streams every ten minutes. The meta-data include that the game name, stream ID, broadcaster's channel ID, current views, created time and other information. Our crawler analyzed these meta files to create the sets of broadcasters' channels and scrape the number of the total views and durations of past broadcasts of each broadcaster every day. Because every past broadcast only counts the number of viewers during its broadcast, the number of total views indeed reflects the characteristics of live streams. Table 2.1 shows the details of the REST APIs used in our crawler. Our dataset includes 2,923 active broadcasters (i.e., sources), who have broadcast a total of 105,117 live performances, attracting over 17.8

[2] `http://dev.twitch.tv/`

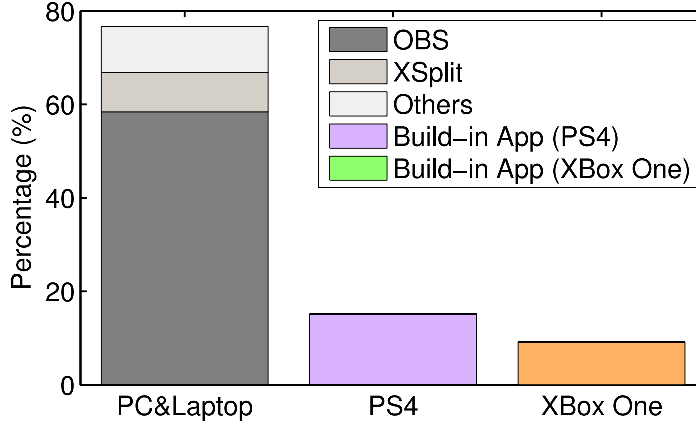


Figure 2.1: Device distribution of Twitch's live sources

million viewers. That is, each source has conducted around 36 live broadcasts in the two-month period. These broadcasts are of different durations and viewer populations, as we will analyze in the next section.

The broadcast sources can be quite heterogeneous, involving PCs, laptops, and even PS4/XBox game consoles, and multiple sources can be involved in one broadcast event. For instance, recent Dota2 game championships "The Summit 2" embraces at least three sources to stream this event, including two game competitive players and a commentator's perspective. Figure 2.1 plots the distribution of the broadcasters' devices in Twitch. Given that the build-in Apps of PS4/Xbox were available just after March 2014, we can clearly see that the PC/Laptop are the most popular devices, at about 76.7%; the second is PS4, at about 15.1%; and the third is XBox One, at about 9.2%. This figure also indicates that the most widely used streaming software on PC/laptop platform is Open Broadcaster Software (OBS) [3], at about 59%.

Our analysis results show that Twitch deploys RTMP (Real Time Messaging Protocol over HTTP Tunnel) streaming servers, covering 14 regions, to compensate the weaknesses of the sources, e.g., networking fluctuation and inferior performance. The original streaming will be transferred through HTTP Live Streaming from streaming servers to viewers with the assistance of a CDN, whereas all the servers are of names: video#.sfo##.hls.twitch.tv, which also indicates the location of the corresponding CDN server, e.g., "sfo" for San Francisco. It is known that Twitch further deploys load balancing servers (usher.twitch.tv) to optimize the live streaming distribution [30] and deliver HTTP Live Streaming playlist file *channelID.m3u8* to each viewer's device. To accommodate heterogeneous viewers, Twitch also provides adaptive online transcoding service to premium content broadcasters. All the live performances can be watched by web browsers or Twitch Apps for mobile devices (e.g.,

[3] `https://obsproject.com`

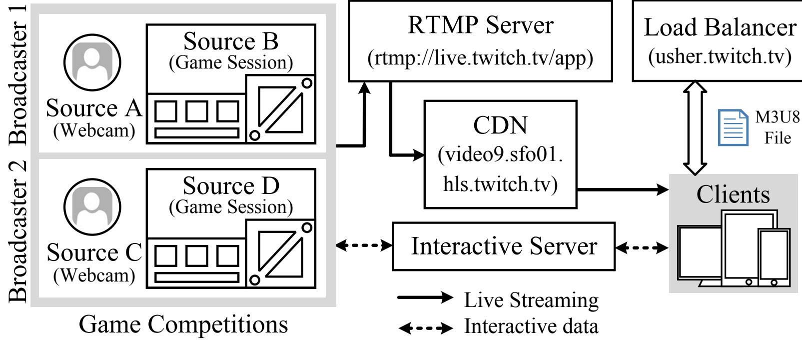


Figure 2.2: Two broadcasters/game players measurement configuration

iOS or Android-based). If a premium broadcaster enables online transcoding service, the browser-based viewers can manually select a proper quality from `Source`, `High`, `Medium`, `Low`, and `Mobile`, and the option `Auto` (i.e., adaptive streaming) is the default setting for a mobile user. However, as we will show, the duration of 50% sessions are over 150 minutes, which imposes too much overhead to transcoding, and hence non-premium broadcasters can only make a tradeoff by selecting a streaming quality for most of the viewers.

Interactive communication is a unique feature in such a Twitch-like crowdsourced system. A set of interactive messaging servers receive the viewer's live messages, and then dispatch the messages to the corresponding live broadcaster and other viewers, enhancing the participants' experience for the live events towards realistic competition environment. That said, the viewers are no longer passive, but can affect the progress of the broadcast as well. In particular, for broadcasting live game playing, the interactive service allows viewers to interact with the game players and commentators in realtime. Our data reveal that these servers for interaction are only deployed in North America using the IRC (Internet Relay Chat) protocol; yet they deliver all the live messages worldwide with reasonably low latency, as we will show in Section 2.4.

To closely investigate the behavior and experience of individual sources and viewers, we also set up three source-end PCs (one commentator and two game players) and five viewers over the Twitch platform, and use network tools, including Wireshark, tcpdump, and ISP lookup, to monitor their detailed incoming and outgoing traffic. Figure 2.2 describes the basic two-player competition broadcast setup for game DotA2. Each player has installed a web camera that captures the video in realtime and encodes in H.264 locally with OBS v0.63b, which is then transmitted to the Twitch platform through RTMP. All devices in our platform are of household PC/tablet configurations, which ensure that our measurement results are representative for general users. The configuration of each device is shown in Table 2.2 and 2.3. The iOS and Android devices were jail-broken/rooted to capture the incoming/outcoming traffic precisely. We also deployed a NETGEAR GS108PEv2 switch to simulate the dynamic uploading bandwidth on the hardware level, which is much more accurate than a software limiter. Finally, to quantify the latencies on the viewer's side and

the impact of network dynamics on Quality-of-Experience (QoE), we use the commentator's laptop (B1) as NTP (Network Time Protocol) server and synchronize other devices to improve the accuracy of measurement results.

Table 2.2: Configuration of broadcasters' devices

ID	Type	Operating Sys.	Uploading
B1 (Commentator)	Laptop	Windows 8.2	2-12 Mb/s
B2 (Players)	Desktop	Windows 8.2	5-18 Mb/s
B3 (Players)	Desktop	Windows 7	3-15 Mb/s

Table 2.3: Configuration of viewers' devices

ID	Network	Operating Sys.	Downloading
P1	Wired	Windows 7	160-250 Mb/s
P2	Wireless	Windows 8.2	7-25 Mb/s
M1	Wireless	iOS 8.0	2-25 Mb/s
M2	Wireless	Android 4.2.2	4-36 Mb/s
M3	3G	Android 4.2.2	0.6-1.2 Mb/s

2.2 View Statistics and Patterns

We analyze Twitch views data and find that it represents several novel and unique characteristics. As of October, 2014, the peak of concurrent streams is above 12000, most of which are for online gaming broadcast. These game streams attract more than one million views every day. We first investigate the characteristic of views in different live contents, and then discuss the source-driven and event-driven views.

2.2.1 Popularity and Duration

The number of viewers is one of the most important characteristics, which reveals the popularity and access patterns of the content. For our global view dataset containing more than 105 thousand streams, we plot the number of views as a function of the rank of the video streams' popularity in Figure 2.3. Clearly, the plot has a long tail on the linear scale; it does not fit a classical Zipf distribution, which is a straight line in a log-log scale, as shown in Figure 2.3. We also plot two other typical distributions, Weibull and Gamma. Because they have heavy tail, especially in the top part, and have been demonstrated to be better fits in YouTube [13], they are also good in the Twitch's case, either. We also calculate the coefficient of determination R^2 to indicate the fitness in this figure. Weibull and Gamma distributions can fit the rise part, in which the popular streams hosted by famous players or commentators attract a large number of game fans through broadcasting game competitions. We also analyze the influences of live events in Section 2.2.2.

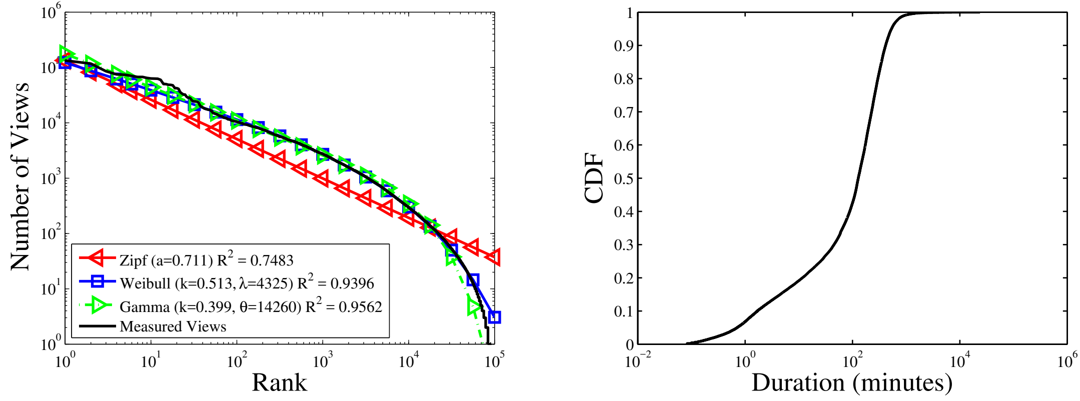
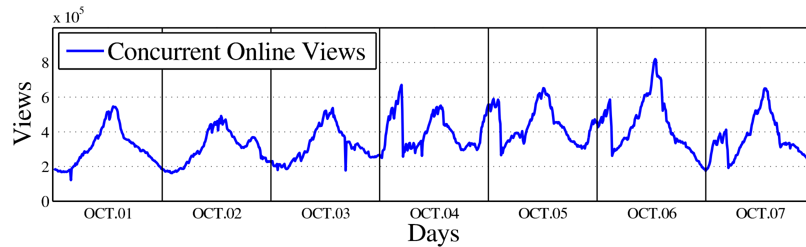
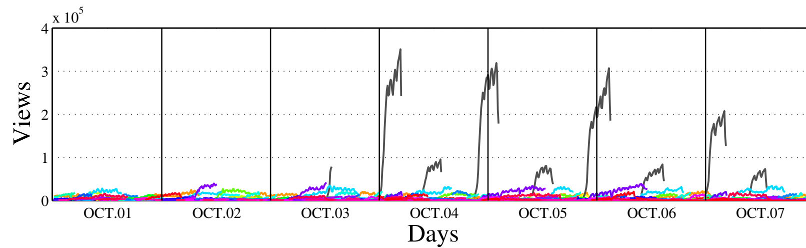


Figure 2.3: Live streams rank ordered by views

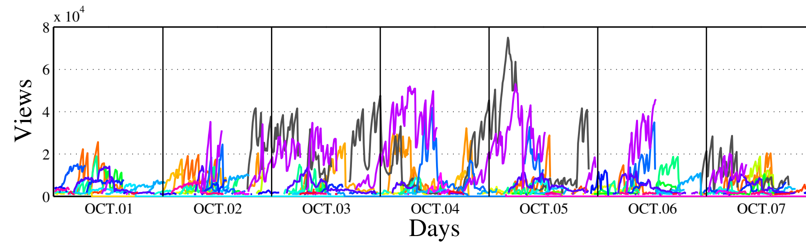
Figure 2.4: Distribution of live streaming duration



(a) Total views



(b) League of Legends



(c) Dota2

Figure 2.5: Views patterns in Twitch (From 2014OCT01 to 2014OCT07)

To understand Twitch's uniqueness, we closely examine the relationship among the total number of views and broadcasters, and the number of views in top broadcasters every ten minutes in our dataset. We find that top-0.5% broadcasters contribute to more than 70% of the total views in general. In several extreme cases, the top-0.4% broadcasters

account for more than 90% of total views. As such, the distribution of the views in Twitch exhibits extreme skewness, being much stronger than conventional streaming systems, e.g., YouTube [87] and PPLive [46].

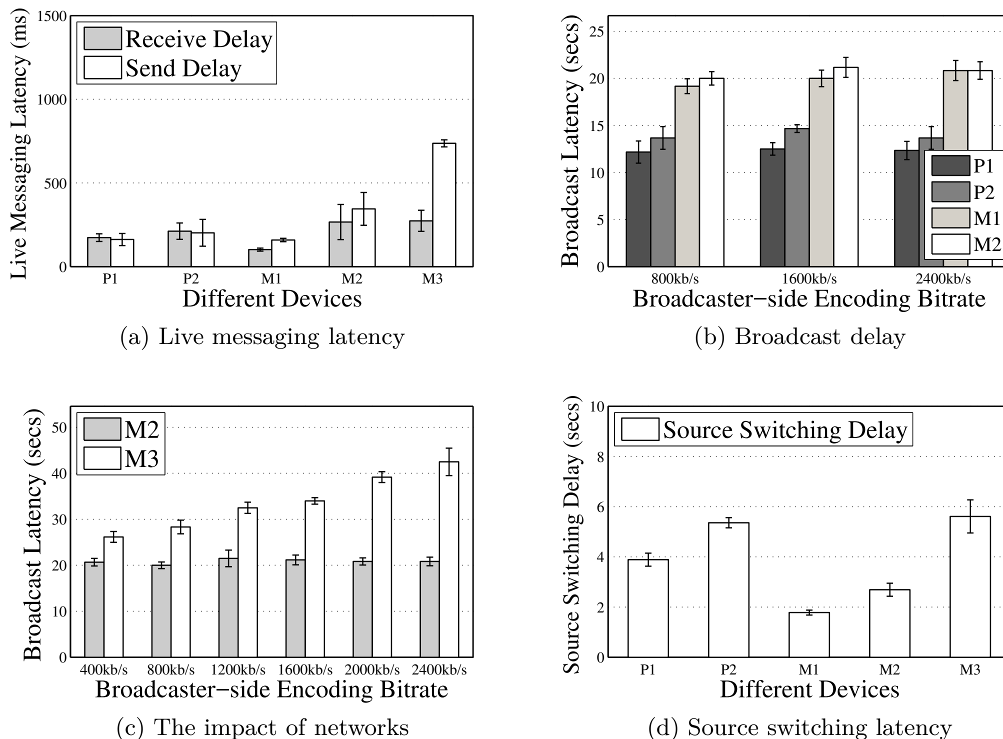
When considering the computation and traffic impacts of a broadcast, popularity is not the only factor, which must be weighted together with the duration of a broadcast. As shown in Figure 2.4, the streaming durations are highly diverse, too. About 30% live contents have a duration around $60 - 120$ minutes, but there are also 30% being more than 4 hours, which is dramatically longer than those in typical user-generated video sharing platforms, e.g., YouTube, where the longest steam is around 2-3 hours (i.e., movies) [13]. The exact duration of a Twitch broadcast, which depends on the interest of and the interaction with the viewers, can hardly be predicted in advance, either. This again is different from professional content services, e.g., TV broadcast. Such long-lived and yet unpredictable broadcast apparently pose challenges on computation and bandwidth resource allocation for real-time online transcoding and reliable broadcasting.

2.2.2 Event- and Source-Driven Views

Due to the globalized demands with time/region diversities, it is well-known video services always experience dynamics and fluctuations requests [10]. To understand the view dynamics of Twitch, Figure 2.5a depicts the online views over time in a one-week period (from OCT01 to OCT07, 2014). The number of concurrent online views exhibits daily patterns: like in the conventional video services [87], the Twitch viewers tend to watch game streaming during the day and evening, whereas less likely in midnight. Interestingly, the number of views was the highest around the midnight on OCT04 and then hastily decreased to the lowest level, implying that if a prominent source can indeed attract massive viewers, despite time. Similar (though less striking) patterns can be seen in OCT05, 06, and 07.

There are also two transient drops from time to time, e.g., on OCT03. After investigating the broadcasters' data, we find that a popular live streaming was disconnected for an unknown reason but re-connected quickly. Accordingly, the number of viewers decreased instantly but managed to recover in a few minutes after re-connection. Such situations rarely happen for professional broadcasters, which have highly reliable equipment setup and network connections. Crowdsourced broadcast system, e.g., Twitch, on the other hand, relies on the non-professionals to provide the broadcast content in realtime; as such, even if the Twitch platform itself is highly reliable with over-provisioned resources, it can hardly guarantee the source video quality.

To further understand the roles of the sources, Figure 2.5b and 2.5c detail the number of views among top broadcasters in two game categories (League of Legends, and Defense of the Ancients 2) during one week. As can be seen, the broadcast can be suspended suddenly; e.g., there are four obvious rises in Figure 2.5b which dropped immediately, due to terminating the game competitions. Since the live progress depends on what is actually happening in the



(a) Live messaging latency

(b) Broadcast delay

(c) The impact of networks

(d) Source switching latency

Figure 2.6: The characteristics at the viewer-side (error bars are 95% confidence intervals)

game competitions, the duration and the exact time of termination can hardly be predicted (see for example the variations in 2.5c). The exact reason and time that trigger a views burst can hardly be predicted, either. Note that these still hold with content other than gaming, as long as they are provided by distributed crowdsources. In short, the views of a crowdsource live streaming system can be more dynamic and unpredictable than conventional video services, and the views are both *event-* and *source-driven*. Even though the Twitch platform is aware of the online status of the massive sources and viewers, significant efforts are still needed to provide the persistently good user experience.

2.3 Messaging and View Latency

We next examine the latencies in the Twitch system, which are critical to the user experience, particularly with live interactions. To this end, we focus on the latencies experienced by a set of representative viewers with typical devices and network settings, namely, wired PC viewer (P1), wireless PC viewer (P2), and mobile tablet viewers (M1, M2, M3). Three latencies are of interest here, namely, *live messaging latency*, *broadcast latency*, and *switching latency*.

2.4 Live Messaging Latency

A distinct feature of the crowdsourced content production is that all viewers and broadcasters can interact and discuss the current live events, which collectively affect the ongoing and upcoming broadcast content. Twitch enables the collaboration via *live messages* exchanged through a set of interactive servers, as shown in Figure 2.2. We capture the networking traffic from five devices and three sources and analyze the send/receive timestamps of live messages (see Section 2 for the experiment configuration). Figure 2.6a presents the live message latencies for five representative viewer devices in our experiments. This type of latency depends on both network conditions and device types; two desktop devices witness the almost same latency between the message sending and receiving operations, whereas the receiving latency of mobile devices is lower than the sending latency. Yet the measurement results suggest that, in general, the live message latency is quite low ($\leq 400ms$), enabling responsive interaction among the participants (viewers and broadcasters). It is worth noting that, along with the live message, the certain control information including a participant's type and streaming quality is also sent to a Twitch statistic server (mp.twitch.tv), as found in our captured network data.

2.4.1 Broadcast Latency

We next measure the *broadcast latency*, which is defined as the time lag of a live event when viewers watch the live streaming from the source. It reflects a viewer's time difference with the commentator and other viewers when they watch and discuss the current live event. A long broadcast latency will obviously affect the interactivity.

Figure 2.6b shows the average, maximum, and minimum broadcast latencies of the four viewer devices (P1, wired PC; P2, wireless PC; M1, M2, mobile tablet). We first vary the streaming bitrates from 800 Kb/s to 2400 Kb/s, and ensure that the downloading bandwidth of each device is significantly higher than the streaming bitrate, so as to mitigate the bottleneck within the network. As shown in the figure, the browser-based P1 and P2 have a latency about 12 seconds for different streaming rates, whereas the client-based M1 and M2 have about 21 seconds. We closely investigate the traffic of each device and find that Twitch adopts a device-dependent broadcast latency strategy to gather the crowdsourced content for processing and to ensure smoothed live streaming. For desktop devices, the inevitable latency derives from that Twitch receives and converts RTMP streaming to HTTP Live Streaming chunks, each of which is a four-second streaming segment; on the other hand, for mobile devices, Twitch will strategically send three more chunks than desktop devices, if all devices start to play live streaming simultaneously. That is, even if we consider an ideal network, mobile devices will still suffer an extra 12 seconds broadcast latency in the current Twitch platform. To evaluate the impact of network bottlenecks, we also compare the latencies of mobile devices with WiFi (M2) and 3G (M3) networks, as shown in Figure 2.6c.

As can be seen, the latencies for M2 remain almost constant across all streaming rates, and M3 incurs extra network delays that increase with the growing streaming rate. The extra network latency is significant only with very high streaming rates, which implies that the processing time (about 10s for all devices) and strategic delivery (extra three chunks for mobile devices) within the Twitch platform are the key factors in broadcast latency.

2.4.2 Source Switching Latency

Given the massive sources available, a viewer has rich choices and can frequently switch among different sources, for the same broadcast event, or even to a totally different event, both of which are now done manually in Twitch (per viewer's action). To investigate the latency of source switching, we record the time duration for 100 switches performed by the different types of devices in different network environments, as shown in Figure 2.6d. Not surprisingly, a higher downloading bandwidth enables a lower switching latency in both wired and wireless networks (e.g., 4 seconds for a high speed wired network and 5.5 seconds for a low-speed wireless network). The latency however is not proportional to the bandwidth; in particularly, the devices in the mobile networks generally have lower switching latencies than those in the wired network, although the mobile bandwidths are indeed much lower, which again indicates different device-dependent strategies have been applied within Twitch.

2.4.3 Impact of Broadcaster's sources

So far, we have examined the latencies on the viewer's side, which includes not only the processing time within the Twitch server and the time from the server to the viewer, as in conventional streaming systems, but also the latency from the source to the server, a new component in the crowdsourced system. Through household Internet accesses and multimedia-ready PCs or mobile devices, anyone can become a Twitch broadcaster, anywhere and anytime. These non-professional broadcasters however have diverse networking connections, both in terms of capacity and stability, especially with wireless mobile accesses. To evaluate the network impact, we deploy a modified OBS module on every broadcaster to record the bandwidth consumption, and first initialize live streaming service in the networks with sufficient uploading bandwidth.

To understand the impact, we next control the maximum uploading bandwidth following five settings: No Limit, 4000 Kb/s, 2000 Kb/s, 1000 Kb/s, and 512 Kb/s; each one lasts five minutes (300 seconds), and the setting finally returns to No Limit at the 1500 second. The original streaming encoding setting is still 4000 Kb/s, and the measurement results are shown in Figure 2.7. From this figure[4], we observe that the number of total dropped fra-

[4]For simplicity, we only show the broadcast latency between P1 and B1. To avoid measurement bias, we repeat the same test on another two broadcasters' devices B2/B3 and other viewer's devices. The results remain consistent with Figure 2.7.

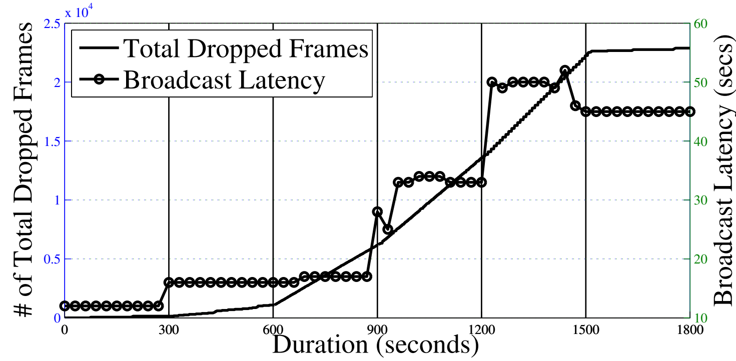


Figure 2.7: Impact of broadcaster's network

mes consistently grows with decreasing the uploading bandwidth on the broadcaster's side. In the meantime, the broadcast latency on the viewer's side also suffers the stepwise rise; in particular, the live streaming experiences two notable delay increases at 900 and 1200 seconds. That said, Twitch attempts to maintain a stable broadcast latency, but cannot guarantee the smooth live streaming. Another interesting phenomenon occurs after recovering the broadcaster's uploading condition (1500-1800 seconds). In this case, the uploading capacity becomes sufficient again, and the broadcaster can offer a stable streaming to Twitch; yet Twitch just decreases the broadcast delay slightly to mitigate the impact of previous networking diversity at the broadcaster-side. These measurement results indicate that the streaming service provided by Twitch is vulnerable and sensitive when the broadcaster's networking capacity is changed frequently, not to mention responsive interactions.

2.5 Summary

In this chapter, we presented an initial investigation on the crowdsourced livecast systems, using Twitch as a case study. Closely examining the access data collected in a two-month period, we outlined the inside architecture of Twitch, and revealed that the views patterns are determined by both the event and the broadcasters' sources. Our measurement also explored the unique source-driven and event-driven views, showing that the current delay strategy on the viewer's side substantially impacts the viewers' QoE, and there is significant inconsistency among the long broadcast latency and the short live messaging latency. On the broadcaster's side, the dynamic uploading capacity is a critical challenge, which noticeably affects the smoothness of live streaming for viewers.

Chapter 3

Cloud-assisted Crowdsourced Livecast

Recent years, crowdsourced livecast has emerged as a powerful and popular streaming service over the Internet [35]. Such livecast services as *Twitch.tv* (or Twitch for short), *GamingLive*[1], and *Dailymotion*[2], allow Internet users to broadcast cooking shows, costume design, music-making, game playthrough[3], etc., attracting an increasing number of viewers around the world. One recent report from Twitch[4] revealed that more than 30,000 broadcasters stream game playthrough on Twitch simultaneously and over 10 billion messages are delivered by its live chatting service a day. To accommodate the growing number of broadcasters and viewers, Twitch is aggressively expanding dedicated servers clusters into high-demand areas[5]. Currently, it has 31 service regions (i.e., ingesting regions) across five continents[6].

To better understand the challenges and opportunities therein, we have closely monitored 1.5 million broadcasters and 9 million streams within one month on Twitch. We find that, despite the success of numerous celebrities, there indeed exist many more broadcasters who have very few or even no viewers. These unpopular broadcasters have irregular schedules, starting or terminating their broadcast programs at any time. They have created highly dynamic workloads to the Twitch's servers and consumed a significant amount of valuable server resources continuously. In particular, over 25% of the bandwidth resources

[1] `www.gaminglive.tv`

[2] `www.dailymotion.com`

[3] When game players play games, they also broadcast the monitor contents from their game devices to fellow viewers with the real-time comments.

[4] `https://goo.gl/cHW1md`

[5] `https://goo.gl/S9mfCS`

[6] `https://twitchstatus.com/`

and over 30% of the computational capacity are used to host the broadcasters with no any viewer at all. These unpopular broadcasters are not only in greater numbers, but also harder to be managed with the irregular schedules and resource consumptions. They are not yet considered in the optimization of existing streaming systems, making the optimal resource allocation quite challenging.

Previous studies have shown the potentials of public clouds in accommodating the dynamic patterns of workloads [56][70]. The technical report from Twitch, however, revealed the weakness of completely employing public clouds in terms of the higher expense, the latency concerns, and the inflexible management[7]. We, therefore, explore the feasibility of using dedicated servers and public clouds cooperatively. Through a series of measurements based on Amazon EC2[8] (EC2 for short) and PlanetLab[9] nodes, we find that public clouds can provide comparable performance in the ingesting and transcoding steps. Yet given the existence of different service regions, how to assign the broadcaster to the regions with minimum operation costs remain challenging.

In this chapter, we present CACL (Cloud-assisted Crowdsourced Livecast), a generic framework that facilitates a cost-effective migration for broadcasters' workloads. In this framework, we first design a stability index (s-index) to characterize a broadcaster's degree of stability in the workload patterns. Then, we formulate and solve the resource allocation problems in ingesting and transcoding steps, considering the diverse capacities and expenses in different regions. Trace-driven evaluations show that our proposed solutions migrate up to 59.9% of workloads from the dedicated servers to the public cloud and reduce about 20% of leasing cost compared with other cloud-assisted strategies.

3.1 Related Work

Some recent studies have already focused on crowdsourced livecast systems. Kaytoue *et al.* [38] introduced the characteristics of Twitch from the perspective of web communities. To address the transcoding problem for non-professional broadcasters, Aparicio-Pardo *et al.* [6] first analyzed the Twitch dataset, and then proposed an optimal model to improve the viewer's satisfaction. Shea *et al.* [62] conducted an empirical performance study and profile the architecture of Twitch. Their work further extended the Twitch framework through bridging cloud gaming platforms and live streaming services. Essaili *et al.* [21] explored the QoE-based uplink resource allocation of user-generated video content. The proposed solution improves resource utilization in mobile networks. Our work differs from these recent studies in the following aspects: first, we focus on how to cost-effectively accommodate the dynamic

[7]`https://goo.gl/tFCxcu`

[8]`https://aws.amazon.com/ec2`

[9]`https://www.planet-lab.org/`

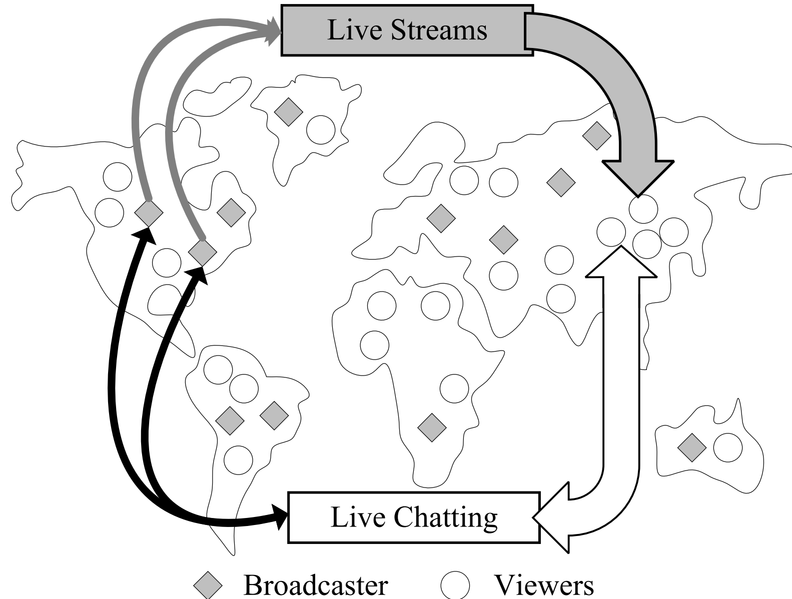


Figure 3.1: A generic system diagram of crowdsourced livecast platforms

and irregular workloads in crowdsourced livecast systems; second, our cloud-assisted design utilizes the flexible resources from public clouds as a complement, with the minimum change to the existing architecture.

On the other hand, cloud transcoding, as a critical component of live streaming, has emerged in the current industrial market. For example, Amazon provides its online transcoding service "Elastic Transcoder (ETS[10])" that works with the master copy of contents in Amazon S3, but does not support the transcoding tasks of live streaming. Another cloud platform Bitmovin[11] supplies both the on-demand and live transcoding service to customers. Similar services also include Zencoder[12], PandaStream[13], EncoderCloud[14], etc. There have been significant researches on cloud-assisted transcoding in recent years. Most of these works examine the characteristics of on-demand video and design the cloud-assisted transcoding architectures in the practical scenarios. Li *et al.* [45] presented "Cloud Transcoder" to transcode the high resolution and heterogeneous videos from mobile devices. Ma *et al.* [51] proposed a scheduling strategy on video transcoding for DASH (Dynamic Adaptive Streaming over HTTP) in a cloud environment through monitoring the workload on each virtual

[10] `http://aws.amazon.com/elastictranscoder/`

[11] `http://www.bitmovin.net/`

[12] `https://zencoder.com/en/`

[13] `https://www.pandastream.com/`

[14] `http://www.encodercloud.com/`

machines. Different from these works, we deploy ingesting and transcoding services on public clouds and optimize the resource allocation for the dynamic broadcasters' workloads in the crowdsourced livecast scenario.

3.2 Measurements of Crowdsourced Livecast: Twitch as a Case Study

In this section, we try to answer the following questions: *how many unpopular broadcasters exist in real crowdsourced livecast systems?* And, *what is the underlying impact of those unpopular broadcasters on the platform performance?* We closely investigate the broadcasters' workloads and the corresponding resource consumptions using the crawled data from Twitch, the largest commercial crowdsourced livecast platform[15].

3.2.1 Twitch-based Datasets

The crawled data are continuously collected from Twitch every five minutes in a one-month period (Feb. 1st-28th, 2015). Through the official APIs, our multi-threaded crawler[16] obtained information from each broadcaster and the official system dashboard[17]. We retrieved both the broadcaster dataset and stream dataset from it[18]. We have excluded certain outliers[19] from the two datasets. A brief explanation is as follows:

- in broadcaster dataset: each trace collects the total number of views and other statistics such as the device type (PC, XBox, or PS4), partner status[20] and the playback bitrate and resolution of source quality, for a total of more than 1.5 million broadcasters (2% outliers have been eliminated).

- in stream dataset: each trace records the number of viewers every five minutes and other properties including the start time, duration, game name, etc., for a total of more than 9 million streams (0.3% outliers have been removed).

[15] `http://marketingland.com/marketers-paying-attention-twitch-202984`

[16] Our multi-threaded crawler does not need Twitch's API client-ID and avoids the limitation for the maximum number of objects to return in each request.

[17] The official system dashboard provides the statistics of current broadcasters, viewers, and games. Link: https://stats.twitchapps.com/

[18] The multi-threaded crawler and data are available at: `https://clivecast.github.io/`

[19] We remove a broadcaster or a stream from the datasets, if its trace is incomplete due to network outage or other connection/terminal problems.

[20] There are two types of broadcasters: partner and non-partner. Twitch enables quality options for partners, whose viewers can select the preferred streaming quality from the source quality (1080p) to 720p, 540p, etc.

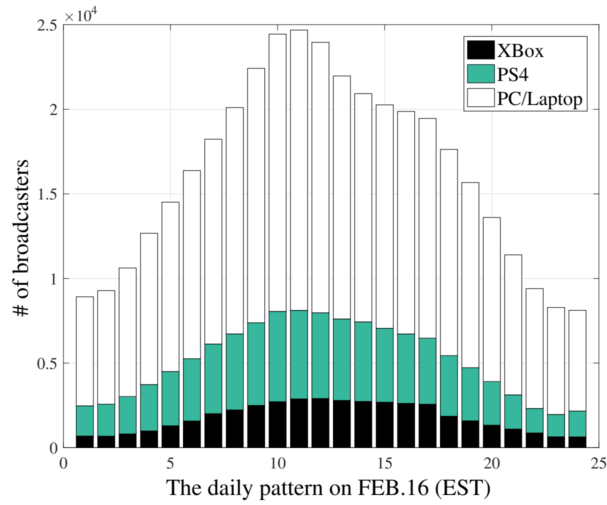


Figure 3.2: Distribution of three types of devices (Eastern Standard Time)

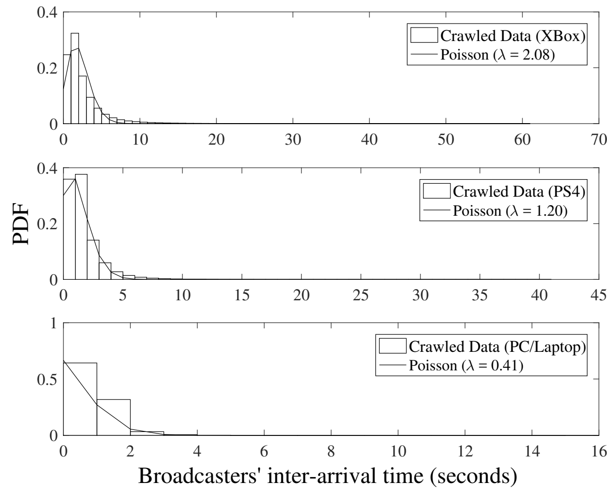


Figure 3.3: Broadcasters' inter-arrival time

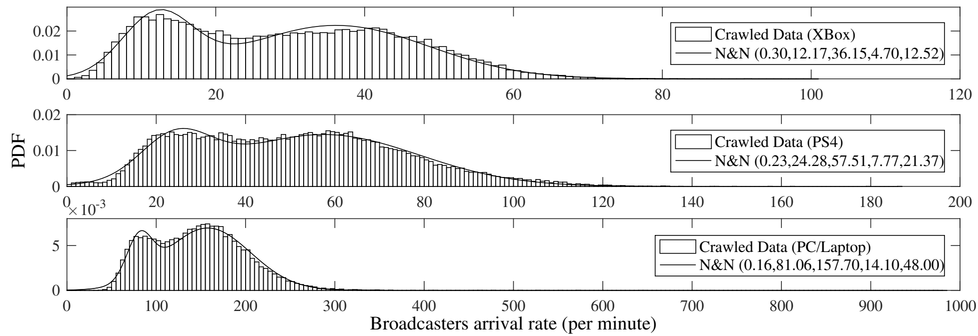


Figure 3.4: Broadcasters' arrival rate

3.2.2 Characteristics of Crowdsourced Live Broadcasters

Broadcasters can stream their game playthroughs from XBox, PS4, and PC/Laptop. XBox and PS4 connect to Twitch's ingesting servers through built-in applications directly, and PC/Laptop captures the live contents from the monitor by various hardware (e.g., Roxio Game Capture HD Pro) or software (e.g., XSplit[21] and OBS). We measure the percentage of each type of devices in the broadcaster dataset. The result shows that: the most popular device is PC/Laptop, at 65%-85%; the second is PS4, at 5%-25%; the third is XBox, at 5%-15%. Figure 3.2 exhibits the proportion of three types of devices during one day.

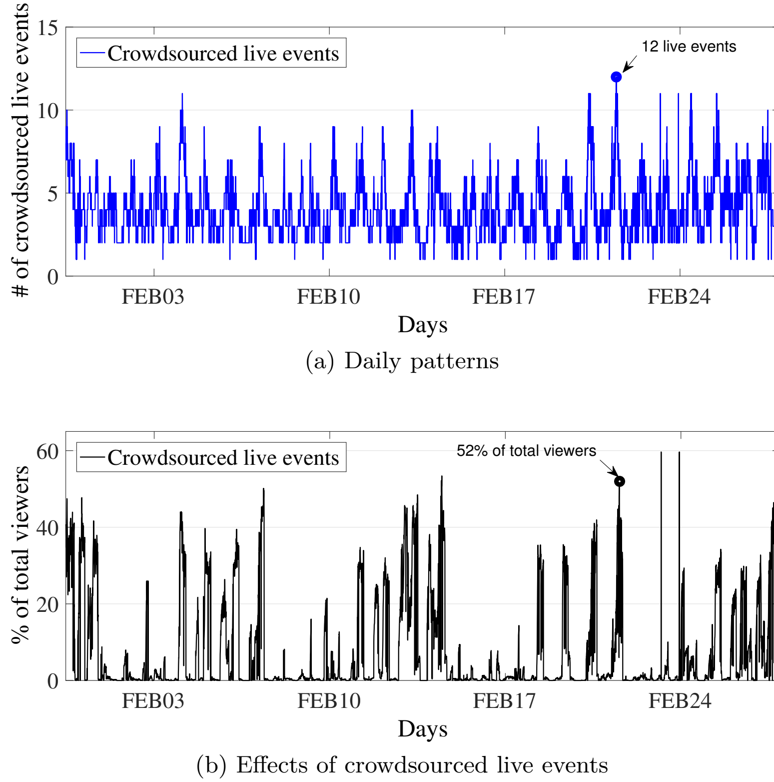
In our broadcaster dataset, we also record the time when each broadcaster starts, so we can closely examine the inter-arrival time and arrival rate of the broadcasters during the one-month period. Figure 3.3 plots the Probability Density Function (PDF) of broadcasters' inter-arrival time. As can be seen, the inter-arrival time of more than 60% of XBox broadcasters is less than two seconds. The percentages are 75% and 90% in PS4 and PC/Laptop, respectively. Note that a Poisson Distribution can be used to fit the inter-arrival time on three platforms with different parameters λ.

Figure 3.4 shows the PDF of broadcasters' arrivals per minute. The crawled data in three types of devices show similar distributions, which exhibit two peaks. These peaks are mainly caused by the daily pattern of broadcasters. In particular, the whole streaming system has the lowest workloads at midnight and the highest at noon; therefore, the first peak is generated by the midnight workloads, and the second peak is caused by the noon workloads in this figure. Besides, this figure also illustrates the different activities of broadcasters in different types of devices. For example, the arrival peaks on the XBox platform are only 12 and 41 arrivals per minute, which is greatly lower than for the other two platforms. This finding also proves the significant disparity of the inter-arrival time in three types of devices in Figure 3.3 as well. The arrival PDF can be fitted by a bimodal distribution, which is a mixture of two normal distributions. The parameters are also shown in Figure 3.4, in which "N&N" indicates that the fitting curve is a component of two normal distributions with parameters $(p, \mu_1, \mu_2, \sigma_1, \sigma_2)$. Parameter p determines the weight of the two normal distributions (i.e., the first normal distribution has a weight p, and the second one has a weight $(1-p)$, $0 < p < 1$). μ_1, μ_2 show the means, and σ_1, σ_2 show the standard deviations.

3.2.3 Effects of Crowdsourced Live Events

Crowdsourced livecast enables event-related live streamings with different broadcasters. For example, five players in one e-sports competition not only cooperatively play a game, but also simultaneously broadcast their game playthroughs to fellow viewers. These streams may be ingested by different streaming servers, and show the distinct contents for this e-sports

[21] `https://www.xsplit.com/`



(a) Daily patterns

(b) Effects of crowdsourced live events

Figure 3.5: Characteristics of crowdsourced live events

competition. These event-related live streams not only have the event-based correlation, but also exhibit the broadcaster-related differences. We first use the broadcasters' names and game types to find these event-related live streams, and then explore their characteristics. Figure 3.5a plots the number of live events during one month. We can find that live events exist in all data traces. Moreover, they attract up to 52% of total viewers in our dataset, as shown in Figure 3.5b. If viewers switch live streams among these broadcasters to select a preferred perspective, the extra latency will impact on the viewers' QoE. As such, we consider the event-related feature in the problem formulation and optimization in Section 3.4.

3.2.4 Popularity of Crowdsourced Live Broadcasters

We then focus on the distribution of broadcaster's popularity, which is a key feature in previous studies for multimedia systems [13][49], and is also critical to answer our first question. We plot the highest number of concurrent viewers against the rank of the broadcasters (in terms of the popularity) in log-log scale in Figure 3.6. From this figure, we observe that

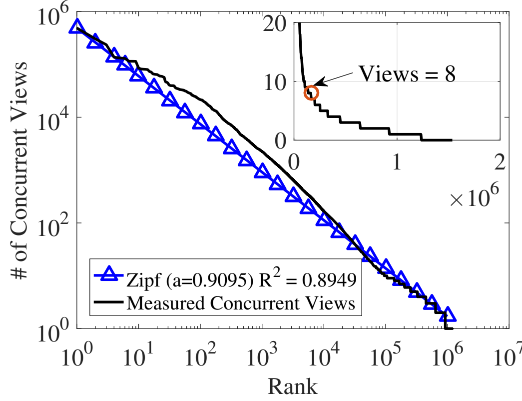


Figure 3.6: Broadcasters rank ordered by popularity

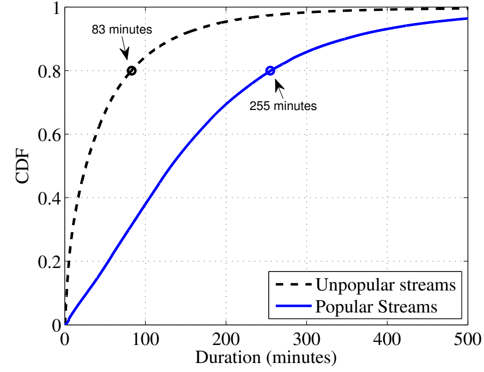
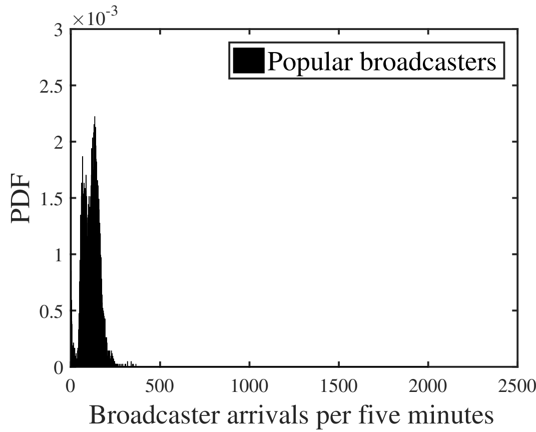
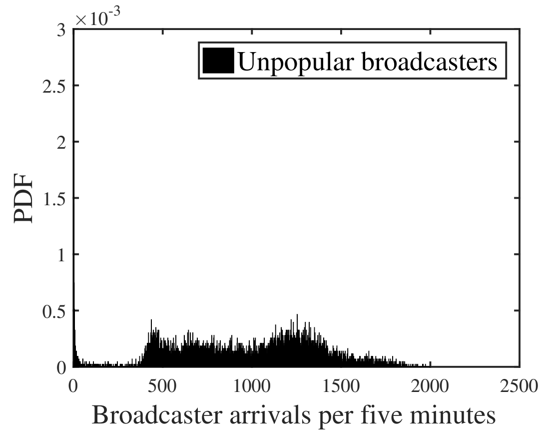


Figure 3.7: Distribution of livecast duration



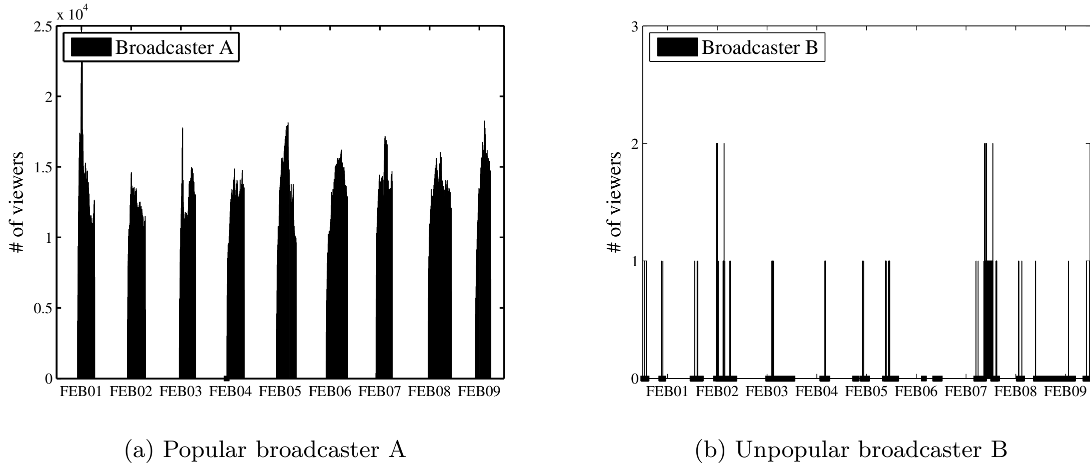
(a) Popular broadcasters



(b) Unpopular broadcasters

Figure 3.8: Broadcaster arrivals per five minutes

the popularity of those broadcasters well exhibits a Zipf's pattern[22]. We further find that there exists such a high skewness, that is, the top-3% popular broadcasters account for about 80% of total viewers at the peak time. Another interesting finding is that 90% of the broadcasters only attract less than 8 viewers (labeled on the small figure in Figure 3.6) even at their peak time. Based on these findings, if the peak number of concurrent viewers in all live streams of a broadcaster is less than 8, we assume that this broadcaster is unpopular and their streams are also unpopular.



(a) Popular broadcaster A

(b) Unpopular broadcaster B

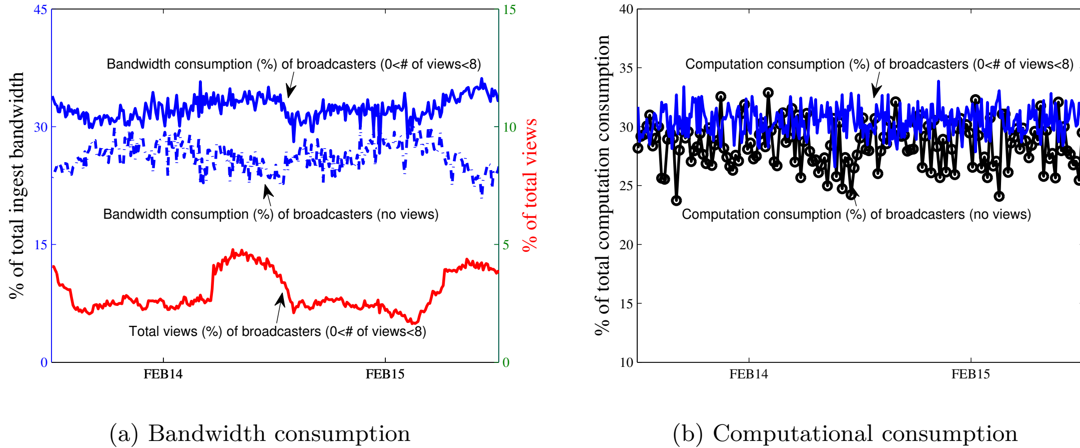
Figure 3.9: Two examples in the broadcaster dataset

3.2.5 Dynamics of Crowdsourced Live Broadcasters

In the stream dataset, the unpopular streams account for 89.5% of all streams. We next try to answer two critical questions: (1) *How long are these unpopular streams?* (2) *Is there any difference between popular and unpopular streams in terms of live duration?* We compare the distribution of their duration with the popular streams, as shown in Figure 3.7. This figure shows that the duration of about 80% of unpopular streams is less than 83 minutes. Because the number of unpopular streams is quite large (about 8.13 million), these unpopular streams could occupy the resources frequently and dynamically in the dedicated servers. We also calculate the total duration of all unpopular streams in one month to be nearly 830 years, while the total duration of popular streams is only 310 years. A huge amount of resources is not utilized effectively.

We also plot the PDF of the broadcasters' arrivals per five minutes in Figure 3.8. This figure shows that the arrivals of the popular broadcasters are clearly lower than 300, while the unpopular broadcasters' arrivals have a considerable range from 400 to 1800. To illustrate the differences between the two types of broadcasters, we plot two typical broadcasters' activities during ten days in Figure 3.9a and 3.9b. Figure 3.9a illustrates that broadcaster A has a regular schedule with a stable live duration, attracting a large number of viewers. Figure 3.9b shows that the broadcaster B attracts a few viewers, but consumes the dedicated resources continuously with the irregular schedule. Due to the frequent arrivals and irregular resource consumption, it is necessary to optimize the dynamic workloads of these unpopular broadcasters in current crowdsourced livecast systems.

[22] We use the coefficient of determination, denoted R^2, to illustrate how well our measured data fit the Zipf's law.



(a) Bandwidth consumption (b) Computational consumption

Figure 3.10: Different types of broadcasters' resource consumptions

3.2.6 Challenges of Hosting Unpopular Broadcasters

To understand the challenges in hosting these unpopular broadcasters, we use the playback bitrate and resolution in the broadcaster dataset to estimate the consumption of bandwidth/computational resources of live streams. The estimation is based on the work in [6], which provides the empirical CPU cycles measurements under different transcoding settings. Figure 3.10 shows the proportion of bandwidth/computational consumption of two types of broadcasters when they stream live content to ingesting servers on Feb 14th/15th, 2015. The broadcasters who do not have any viewers consume about 25% of bandwidth resources and 28% of computational resources. In the meantime, about 33% of bandwidth resources and 31% of computational resources are consumed by the broadcasters who only have less than 8 concurrent viewers. Note that these broadcasters only attract less than 5% of online viewers.

3.3 CACL: Architecture and Design

The results from the Twitch-based measurement have illustrated that the dedicated resources are not utilized effectively and motivated us to design a new crowdsourced livecast framework. In this section, we first examine the feasibility of migrating certain workload to public clouds through an EC2-based measurement, and then present the architecture of our Cloud-assisted Crowdsourced Livecast (CACL) design, which targets on mitigating the impact of current dynamic, unpredictable, and irregular workloads cost-effectively.

3.3.1 EC2-based measurement

Due to the elastic resource provisioning and cost-effective scaling, public clouds have been proven to be an effective complement of dedicated servers for streaming services [1]. For

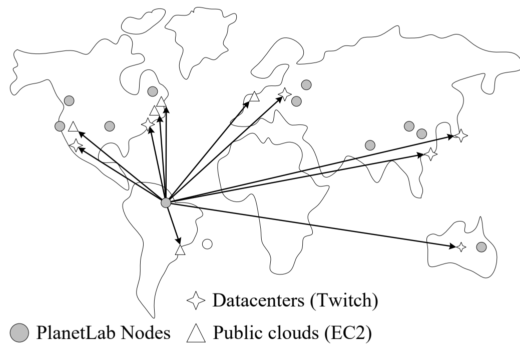


Figure 3.11: Diagram of the EC2-based measurement

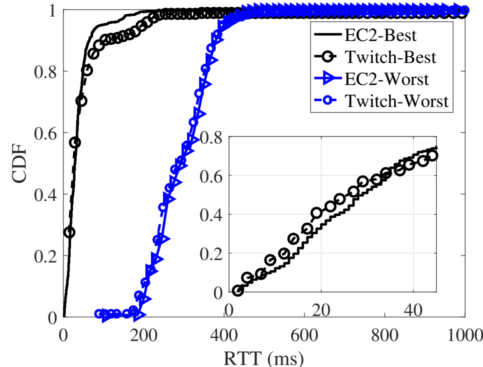


Figure 3.12: RTT comparison

instance, Netflix, the major streaming provider in America, has migrated its streaming infrastructures to Amazon EC2 (EC2 for short) and the storage of master film copies to Amazon S3 (Simple Storage Service) since 2010. For crowdsourced livecast, it remains to identify which workloads to be migrated to the public cloud without sacrificing the QoE of viewers. We next conduct the measurements to investigate this problem based on Amazon EC2 and PlanetLab nodes. We focus on the Round-Trip Time (RTT) between broadcasters and ingesting servers because this metric is mainly used to test the ingesting performance in the broadcasting software (e.g., OBS).

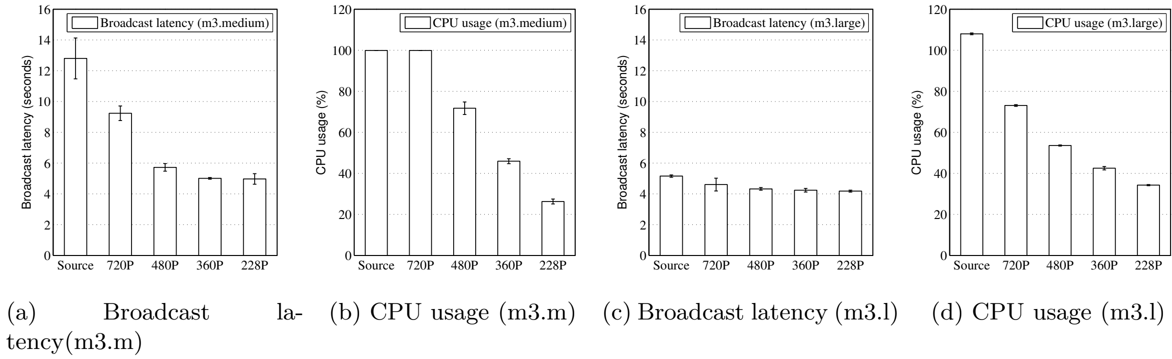
To compare the ingesting performance (i.e., RTT) between the dedicated servers and the public clouds, we deploy eight ingesting servers on EC2 using m3.medium instances with Ubuntu 14.04 and Nginx-RTMP module[23]. Similar to Twitch's dedicated ingesting servers, these EC2 instances, which are located at eight locations (Virginia, Tokyo, Ireland, etc.), can receive/transcode live streams using Nginx-RTMP module and deliver them to geo-distributed viewers. We also set up 224 PlanetLab nodes (the maximum number of available nodes during our study) to run as the broadcasters and measure the performance of the ingesting connection between these broadcasters and ingesting servers, as shown in Figure 3.11. We measure the RTTs between 224 PlanetLab nodes and 26 ingesting servers (18 in Twitch[24] and 8 on EC2) and acquire the following results.

3.3.2 Round-trip Time

Figure 3.12 shows the RTT comparison for the Twitch and EC2 cases. We can observe that about 60% of broadcasters in the Twitch-best case have a quite low RTT (less than 35 ms), but the disparity between the two best cases is quite small, as shown in the small figure

[23] `https://github.com/arut/nginx-rtmp-module`

[24] Twitch had only 18 ingesting regions during our study.



(a) Broadcast latency(m3.m) (b) CPU usage (m3.m) (c) Broadcast latency (m3.l) (d) CPU usage (m3.l)

Figure 3.13: Broadcast latency and CPU usage on different instances

in Figure 3.12. Moreover, nearly 40% of broadcasters can enjoy a lower connection latency (with the maximum up to 150ms) when they choose EC2 instances as the preferred ingesting servers. We also compare the worst cases and find that the RTTs in the EC2-worst case are very similar to the results in the Twitch-worst case, which means that EC2 instances do not increase RTTs significantly even in the worst situations. We, therefore, can use EC2 instances to ingest broadcasters' live streams with the comparable performance. Note that the ingesting step is only the first part of livecast services, we still need to consider the *broadcast latency*, which reflects the viewers' QoE directly.

3.3.3 Broadcast Latency

Our previous work defined two types of latencies in crowdsourced livecast platforms [79]: (1) *broadcast latency*: the time lag of a live event when viewers watch the live streaming from the source. (2) *live messaging latency*: the time difference when a message is sent from a viewer to other viewers. The viewers are more sensitive to broadcast latency, because the disparity between broadcast latency and live messaging latency will affect viewers' QoE in terms of the participation and discussion.

To measure the broadcast latency on public clouds, we lease two types of instances (m3.medium and m3.large[25]) from Amazon's Oregon data-center. We deploy a PC (Dell 7010) with OBS as the broadcaster's device and a laptop (Samsung NP355V5C) with VLC[26] as the viewer's device in a campus network. We stream the active window of a stopwatch application from the broadcaster's PC to the instance and play this live stream on the viewer's laptop. To calculate the time difference, i.e., broadcast latency, between them, we

[25]We lease on-demand instances. The configuration and price are m3.medium: 1vCPU, 2.5GHz, Xeon E5-2670v2, 3.75G memory, $0.067/h; m3.large: 2vCPUs, 2.5GHz, Xeon E5-2670v2, 7.5G memory, $0.133/h

[26]VLC is a free and open source multimedia player and framework, `http://www.videolan.org/vlc/index.html`

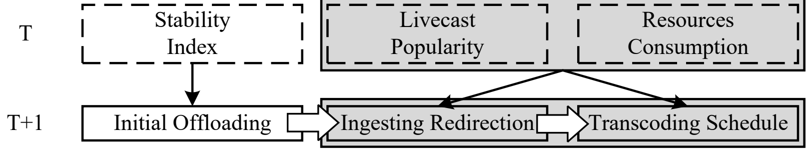


Figure 3.14: Framework of Cloud-assisted Crowdsourced Livecast (CACL)

set up a camera to record two monitors at the same time. The transcoding settings are Source quality (i.e., 1080p, 3200kbps), 720p (1500kbps), 480p (800kbps), 360p (500kbps), and 228p (200kbps). We plot the results in Figure 3.13 with the average values and standard deviations. Figure 3.13a and 3.13b show the broadcast latency and CPU usage on the m3.medium instance in different transcoding settings. We can observe that this instance cannot transcode the source RTMP stream to 1080p and 720p HLS streams due to the overloaded CPU. Yet it can process another three workloads very well and acquire the lower broadcast latency (about 5 seconds) than Twitch, which suffers from more than 10 seconds broadcast latency during live events [79]. Because another m3.large instance has two vCPUs[27], it has better performance, as shown in Figure 3.13c and 3.13d. From Figure 3.13c, we observe that all broadcast latencies of various settings are decreased to about 5 seconds with the sufficient computational capacity. From Figure 3.13d, we find that only the 1080p transcoding task uses the computational resources of more than one vCPU. In summary, the transcoding workloads can be migrated from dedicated servers to public clouds, provided that the instances are carefully selected without increasing the broadcast latency.

3.3.4 CACL Architecture

The livecast broadcasters constantly utilize the streaming service, any interruption will remarkably affect viewer's QoE. Besides, crowdsourced live events, in which several broadcasters simultaneously start live streams, have a more stringent restrictions on broadcast latency.

To overcome these challenges, our design aims to systematically optimize the following three steps, as shown in Figure 3.14: (1) *Initial Offloading*, for the broadcasters who already have historical activities, including the duration and schedule information of live streams, the system assigns an ingesting region to them from public clouds or dedicated servers according to their stability index. (2) *Ingesting Redirection*, based on the broadcasters' popularity, the system allocates a proper ingesting area and redirects the broadcasters' workloads; (3) *Transcoding Schedule*, the system considers the broadcasters' resource consumption and the transcoding capacities in different service regions during the workload migration. Step 2 and 3 have to be designed together, because once a broadcaster's wor-

[27]Each of vCPUs is a hyperthread of an Intel Xeon core

kload is offloaded to a certain ingesting region, the corresponding transcoding workload has to be processed in the same region to reduce the broadcast latency.

3.3.5 Initial Offloading

In the CACL framework, the first challenge is how to allocate a proper ingesting server to the broadcasters at the beginning of live-broadcast. We introduce a stability index (s-index) to calculate a broadcaster's degree of stability: an s-index close to zero means the broadcaster is highly dynamic and close to 1 means it is likely stable and has a regular broadcasting schedule. The s-index of a broadcaster depends on her schedule in recent several days. For example, *Alice* broadcasts her game sessions from 1:00 PM to 3:00 PM in recent three days, while *Bob* has only one stream at the same days; therefore, *Alice*'s s-index is higher than *Bob*'s. For one broadcaster b who has activities in recent n days ($n \geq 2$), we first divide the ith day to m equal time slots, each time slot j has a value $d_{i,j}$ is a binary variable that indicates whether b has a live stream in current time slot. As such, we can use $SI^{(b)}$ to check whether the broadcaster b regularly consumes the bandwidth/computational resources in recent n days.

$$SI^{(b)} = \begin{cases} \frac{1}{n} \sum_{i=2}^{n} \frac{\sum_{j=1}^{m} d_{i,j}^{(b)} \cdot d_{i-1,j}^{(b)}}{\sum_{j=1}^{m} d_{i-1,j}^{(b)}} & \text{if } \sum_{j=1}^{m} d_{i-1,j}^{(b)} \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Given the s-index $SI^{(b)}$ of a broadcaster b, a straightforward way to give the offloading decision is to set a threshold H: if $SI^{(b)} \geq H$, broadcaster b will be assigned to the ingesting servers in dedicated servers, otherwise, to public clouds. Using a firm threshold, however, suffers from the following drawback: if the dedicated servers have a massive amount of spare resources, leasing the instances on public clouds to specifically ingest unpopular workloads will not be cost-effective. We solve this problem by updating the value of H to the average of existing broadcasters' SI per time slot. Followed by the growth of broadcasters, more and more stable broadcasters will be ingested into dedicated servers, and the dynamic broadcasters are offloaded to public clouds.

3.4 Problem Formulation and Solution

Due to the significance of broadcast latency for viewers' QoE, there have been lots of studies on latency minimization for the conventional streaming system, mainly focusing on the transcoding efficiency inside the transcoding servers [45][51][6]. Nevertheless, the latency of user's interaction in crowdsourced livecast systems poses a stringent constraint in the ingesting and transcoding stages of live streams, not to mention the interactions in the live event. Considering the crowdsourced live events and latency disparity, we take the decrease of broadcast latency as our objective and propose a formal description of this

optimization problem. Because the broadcasters are highly dynamic, our design is based on the broadcaster's popularity in real-time.

3.4.1 Basic Model with Ingesting Latency

We first focus on a basic model to optimize the ingesting latency in our CACL framework cost-effectively. We target on maximizing the reduction of latency, when the ingesting region is determined. To make the problem easy to discuss, we quantize time into discrete slots, which may be a few minutes to several hours (e.g., five minutes in our experiment). We use $B^{(t)}$ to denote the set of broadcasters and $E^{(t)}$ to denote the set of crowdsourced live events in time slot t. ($\forall i = 1, 2, \cdots, m, \forall j = 1, 2, \cdots, m$, $e_i \subseteq E^{(t)}$, $|e_i| \geq 1$, $e_i \cap_{i \neq j} e_j = \varnothing$, and $\cup e_i = B^{(t)}$). We define R as the set of ingesting areas where a broadcaster can be connected to upload live contents and define set $W_r^{(t)}$ as the bandwidth demand of ingesting area r. We assume that the instance in public cloud areas are homogeneous and let $\mathcal{W}$ denote the bandwidth capacity of each instance, assuming the intra-area workload allocation is optimized [75][11][70]. We define $L_{(b,r)}^{(t)}$ as the broadcast latency if b selects ingesting area r. It can be calculated as:

$$L_{(b,r)}^{(t)} = l_{(b,r)}^{(t)} + l_r^{(t)} + l_{(r,v)} \tag{3.2}$$

where $l_{(b,r)}^{(t)}$ is the link latency between b and r, $l_r^{(t)}$ is the ingesting latency that is determined by the instance type in r, and $l_{(r,v)}$ is the latency between ingesting server to a class of viewers v. We aim to decrease $L_{(b,r)}^{(t)}$ for each broadcaster cost-effectively.

To fulfill this target, we have to find an assignment $A^{(t)}$ that determines the mapping from B to R in time slot t. We define a utility function $U^{(t)}(b, r)$ that indicates the effects of $L_{(b,r)}^{(t)}$ when b uploads live streaming to ingesting area r. In particular, $U^{(t)}(b, r)$ can be calculated as follows:

$$U^{(t)}(b, r) = G^{(t)}(b, r) \cdot N_b^{(t)} \tag{3.3}$$

where $N_b^{(t)}$ is the number of viewers who watch broadcaster b's live streaming in time slot t. $G^{(t)}(b, r)$ refers to the gain of latency decreasing. Without loss of generality, we assume $G^{(t)}(b, r)$ is a non-negative, strictly concave, and twice continuously differentiable function. The conventional choice is logarithmic function [40], we define $G^{(t)}(b, r)$ as follows:

$$G^{(t)}(b, r) = \alpha + ln(1 - \beta L_{(b,r)}^{(t)}) \tag{3.4}$$

where α and β are two tunable parameters, which control the function shape. We employ $ln(1 - \cdot)$ to make sure that the less the broadcast latency, the more the gain.

Based on the previous definitions, let $I(r)$ be the indicator function which takes value 1 when area r belongs to the public cloud and value 0 otherwise. Given the broadcast latency

between b and r, our objective is to find an assignment A that can maximize the minimum utility $F(A^{(t)})$ among all broadcasters in a live event.

$$\underset{e \in E^{(t)}}{\text{Maximize}} \; F(A^{(t)}) = \min_{\substack{b \in e \\ r \in R}} \{U^{(t)}(b, r)\} \tag{3.5}$$

subject to:

Bandwidth Availability Constraint:

$$\forall r \in R, W_r^{(t)} \leq \mathbb{W}_r \tag{3.6}$$

Bandwidth Cost Constraint:

$$\sum_{r \in R} \frac{W_r^{(t)}}{\mathcal{W}} \cdot Cost_w(r) \cdot I(r) \leq K_w \tag{3.7}$$

where $\mathbb{W}_r$ is the bandwidth capacity of ingesting area r. $Cost_w(r_i)$ is the bandwidth price in the area r_i. The bandwidth availability constraint (3.6) asks that at any given time, the bandwidth demands have to be satisfied. The total budget constraint (3.7) asks that at any given time, the total cost of leasing instances does not surplus total budget K_w.

3.4.2 Enhanced Model with Transcoding Latency

We now extend our model by considering the transcoding workloads in different ingesting areas. Similar to the definition of the previous problem, the objective is to optimize the broadcast latency in the ingesting service regions. Yet we re-define $L_{(b,r)}^{(t)}$ in the equation (3.8), considering the transcoding step with multi-quality streams. For example, Twitch provides five streaming quality options (Source, High, Medium, Low, and Mobile) to viewers. We define V as the set of streaming quality.

$$L_{(b,r,v)}^{(t)} = l_{(b,r)}^{(t)} + l_{(q_b,q_v)}^{(t)} + l_{(r,v)} \tag{3.8}$$

where q_b is the quality (i.e., bitrate) of b's source streaming, q_v is the quality of target version v ($v \in V$). $l_{(q_b,q_v)}^{(t)}$ is the transcoding latency, which can be measured in advance.

We now extend utility function $U^{(t)}(b, r)$ as:

$$U^{(t)}(b, r) = \sum_{v \in V} G^{(t)}(b, r, v) \cdot N_{(b,v)}^{(t)} \tag{3.9}$$

where $N_{(b,v)}^{(t)}$ is the number of viewers who watch b's v version streaming in this time slot. This value is initially determined by b's historical distribution of different versions. $G^{(t)}(b, r, v)$ means the gain when b select r as the ingesting and transcoding area and is calculated as

follows:

$$\begin{aligned} G^{(t)}(b,r,v) &= \alpha + ln(1-\beta L^{(t)}_{(b,r,v)}) \\ &= \alpha + ln(1-\beta(l^{(t)}_{(b,r)} + l^{(t)}_{(q_b,q_v)} + l^{(t)}_{(r,v)})) \end{aligned} \tag{3.10}$$

where $l^{(t)}_{(q_b,q_v)}$ denotes the transcoding latency. If the original quality q_b is no more than the target quality q_v, the transcoding servers only transcode the original RTMP stream to the HTTP-based stream, using the same resolution and bitrate settings. That is, the transcoding latency $l^{(t)}_{(q_b,q_v)} = l^{(t)}_{(q_b,q_b)}$. The transcoding latency depends on the current computing capacity of area r and monotonously increases based on both q_b and q_v [74].

Our objective is extended to a new version as:

$$\underset{e \in E^{(t)}}{\text{Maximize}}\ F(A^{(t)}) = \min_{\substack{b \in e \\ r \in R}} \{U^{(t)}(b,r)\} \tag{3.11}$$

subject to:

Previous Constraints: (3.6), (3.7)

Computational Availability Constraint:

$$\forall r \in R, C^{(t)}_r \leq \mathbb{C}_r \tag{3.12}$$

Computational Cost Constraint:

$$\sum_{r \in R} \frac{C^{(t)}_r}{\mathcal{C}} \cdot Cost_c(r) \cdot I(r) \leq K_c \tag{3.13}$$

where $C^{(t)}_r$ is the computing demand of area r, $\mathcal{C}$ denotes the computing capacity of an instance. $Cost_c(r)$ is the price of an instance in r in terms of computing capacity. The computing availability constraint (3.12) guarantees that at any given time t, the consumption of computational resources in each transcoding task can be satisfied. The budget constraint (3.13) guarantees that the computational cost is lower than the budget K_c, which we assume can at least serve all offloading workloads.

3.4.3 Solution

The objective function (3.11) has four constraints (3.6) (3.7) (3.12), and (3.13), the bandwidth cost and computational cost are not independent due to the pricing criteria of instances on public clouds. Previous studies on EC2 instances already reveal that the bandwidth capacity is more than 700Mbps on m3.large instance [24]. Moreover, our measurement results in Section 3.3.1 also reveal that generating low-latency live streams will consume a vast amount of computational resources. If we relax constraints (3.6) and (3.7), other constraints can still work for the optimization objective function (3.11). Assuming that the capacities of the different service areas are given, our assignment problem can be transformed into a

0-1 Multiple Knapsack problem with a non-linear objective function, which is known to be NP-hard [19].

We thus propose a heuristic solution, which includes two steps: scale decrease and resource allocation. In the first step, as shown in Algorithm 1, we aim to eliminate the redundant assignments based on the optimization target of maximizing the minimum utility in live events. We first get the maximum value from the set of the minimum utility of each assignment (b, r) in crowdsourced live events (line 1-8). We then remove most parts of the solution space (line 9-14) to improve the search efficiency. In the second step, as shown in Algorithm 2, $c(b)$ denotes the computational consumption of transcoding workloads b, we define the new utility $u^{(t)}(b, r)$ of each broadcaster in all events using the equation (3.14) and find the area r^* in the equation (3.15), which is derived from [15]. Then, we sort them in decreasing order of $u^{(t)}(b, r^*)$, which allows the assignment with the higher resource utilization being explored first. According to the sorted broadcasters, we assign them to available service areas.

$$u^{(t)}(b, r) = \frac{C_r^{(t)} \cdot U^{(t)}(b, r)}{c(b)} \tag{3.14}$$

$$r_b^* = \operatorname*{argmin}_{r \in R} \{u^{(t)}(b, r)\} \tag{3.15}$$

Algorithm 1: ScaleDecrease()

1 **for** *each live event* $e \subseteq E$ **do**
2     $U_1^{(t)} \leftarrow \emptyset$
3   **for** *each region* $r \in R$ **do**
4         $U_2^{(t)} \leftarrow \emptyset$
5   **for** *each broadcaster* $b \in e$ **do**
6   add $U^{(t)}(b, r)$ into set $U_2^{(t)}$
7   add $\min\{U_2\}$ into set $U_1^{(t)}$
8     $U_e^{(t)} \leftarrow \max\{U_1^{(t)}\}$
9 **for** *live event* $e \subseteq E$ **do**
10   **for** *each region* $r \in R$ **do**
11   **for** *each broadcaster* $b \in e$ **do**
12   **if** *(*$U^{(t)}(b, r) < U_e^{(t)}$*) and (*$isPath(b) > 1$*)* **then**
13   // $isPath(b)$ returns the number of b's assignments in $A^{(t)}$
14                 $A^{(t)} \leftarrow A^{(t)} - (b, r)$
15   // Remove this assignment from $A^{(t)}$

Algorithm 2: ResourceAllocation()

```
1  for each broadcaster's assignment (b, r) in A^(t) do
2      add u^(t)(b, r) into u_b^(t)
3      // Calculate u^(t)(b, r) using the equation (3.14)
4  B_sorted ← Sorted broadcasters in descendant order of u^(t)(b, r_b^*)
5  // Get r_b^* from u_b^(t) according to the equation (3.15)
6  for broadcaster b ∈ B_sorted do
7      r_sorted ← Sorted available area r of b in descendant order of u_b^(t)
8      for region r ∈ r_sorted do
9          if C_r^(t) − c(b) ≥ 0 then
10             A^(t) ← A^(t) − (b, ·)
11             // Remove all assignment of b
12             C_r^(t) ← C_r^(t) + c(b)
13             A^(t) ← A^(t) + (b, r)
14 return A^(t)
```

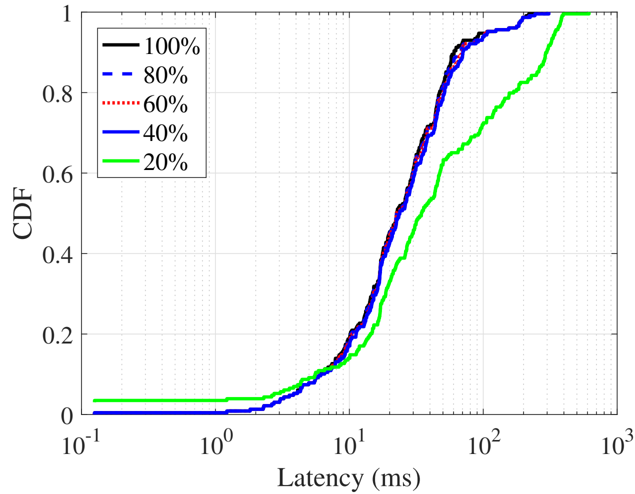


Figure 3.15: Impact of computational capacity

3.5 Performance Evaluation

In this section, we conduct the trace-driven simulations and examine the performance of our cloud-assisted crowdsourced livecast with the proposed algorithms.

3.5.1 Efficiency of Resource Allocation

We first evaluate the performance of our resource allocation algorithm using the traces in the EC2-based measurement (Section 3.3.1). We used the measured RTTs of each PlanetLab node to evaluate the proposed algorithms. Because we already have the distribution of

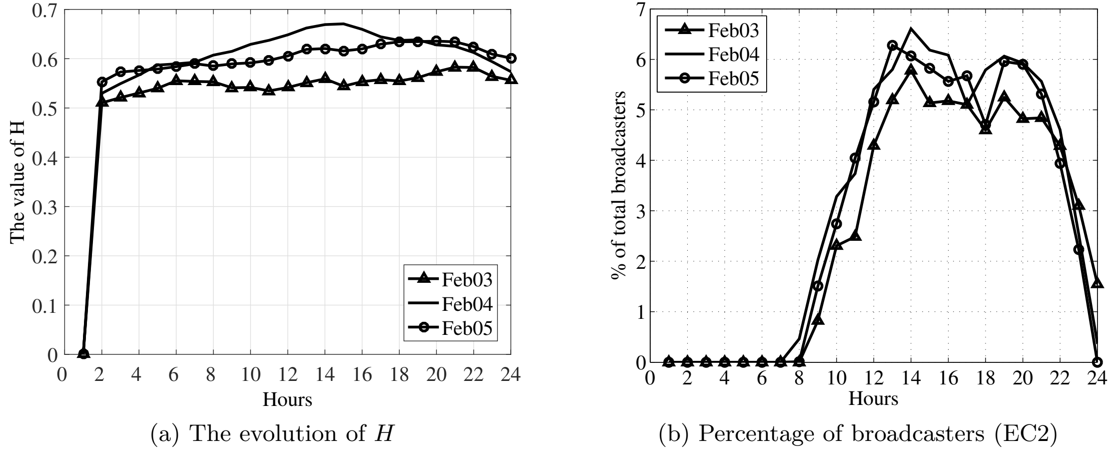
resolution and bitrate according to the Twitch datasets, we can assign the resolution and bitrate settings to every PlanetLab node as a broadcaster and add it into a live event. In the meantime, we set different arrival times and leave times to each node based on the measurement results in the Twitch datasets. We randomly assign 224 PlanetLab nodes (i.e., broadcasters) into 100 crowdsourced events and set the number of viewers and the resolution of every broadcaster according to the Twitch-based measurement in Section 3.2. We assume that all service areas in dedicated servers and public clouds have the same computational capacity. The consumption of computational resources is estimated according to the measurement in [6]. To clearly demonstrate the effectiveness of our solution, we adjust the computational capacity from 100% to 20%. Figure 3.15 demonstrates the impact of different settings in the computational capacity. From this figure, we can observe that a small proportion of broadcasters suffers higher RTTs in 80%, 60%, and 40% of computational capacity. Because 20% of computational capacity is less than the requirement of transcoding all streams from the broadcasters, we can find a significant rise in the ingesting performance. As such, our resource allocation algorithm achieves a similar result with a lower amount of total computational resources.

3.5.2 Trace-driven Simulation

We then conduct the trace-driven simulation based on our Twitch datasets. We make a few simplifications in the simulation based on realistic settings: first, as transcoding consumes most of the computational resources from the instances on public clouds, as shown in our EC2-based measurement, we use the computational resources as the constraint to decide the assignment strategy; second, we consider that the EC2 instances are homogeneous and latency $l_{(r,v)}$ is fixed for a certain quality level of HTTP Live Streaming; third, we ignore the cost in dedicated servers and focus on the cost when workloads are offloaded into the instances in public clouds. The following default settings are used in the simulation: because the broadcast latency[28] in Twitch is from 10 to 40 seconds [79], we set $\alpha = 1$ and $\beta = 0.011$ to make $G^{(t)}(\cdot) \in [0, 1]$ when the broadcast latency $L^{(t)}_{(\cdot)} \in [0, 57]$. We assume that the instance type on public clouds is m3.large based on the EC2-based measurement in Section 3.3.1. The algorithms are launched every five minutes, which also is the time slot of crawling data.

We first study the impact of stability index SI and threshold H. To accelerate the simulation, we calculate the stability index for each broadcaster and save the results in advance. The simulation program can directly acquire the stability index of broadcasters when they start live streams. According to our design, the parameter n is more than or equal to 2. The default setting of n is 2 in our trace-driven simulation. We set the initial threshold

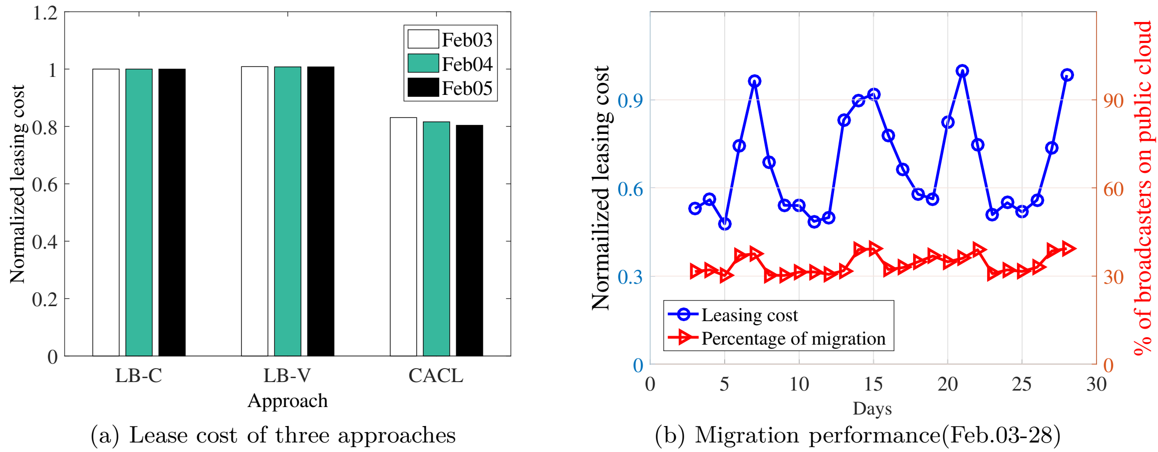
[28] Readers can check your broadcast latency through activating "Show video stats" after clicking Options button when you watch any live stream from Twitch.



(a) The evolution of H

(b) Percentage of broadcasters (EC2)

Figure 3.16: Impact of threshold H



(a) Lease cost of three approaches

(b) Migration performance(Feb.03-28)

Figure 3.17: Performance evaluation of proposed solutions

$H = 0$ and use it to classify the new broadcasters without any other strategies. We assume that the offloading starts when the bandwidth consumption is up to 60% of dedicated severs. Figure 3.16 illustrates the evolution of H and its impact for the public cloud during three days (Feb 3rd-5th, 2015). From Figure 3.16a, we observe that the value of H increases dramatically at the beginning of that day, and then it stabilizes between 0.5 and 0.7. At the peak traffic time (from 9:00AM to 13:00PM), a vast majority of the broadcasters arrive at the streaming system; therefore, the value of H experiences a small decrease. However, the limitation of H induces that the public cloud only hosts a small number (maximum 6.5%) of broadcasters. Thus, threshold H plays a beneficial role on the offloading process, but other strategies, i.e., Ingesting Redirection and Transcoding Schedule, are still needed to reduce the impact of broadcasters' dynamics.

With the previous parameter setting of H, we then conduct the extended simulation to investigate how CACL perform with the real-world data traces. We also propose the views-based (LB-V) migration and computation-based (LB-C) migration as two baseline approaches for comparisons. The LB-V approach migrates the unpopular live stream to the public cloud, considering the number of online viewers. While the LB-C approach migrates workloads to the dedicated servers when the computational resources are still available. Figure 3.17a compares the leasing cost of three workload provisioning approaches: LB-V, LB-C, and CACL-based approaches in three days. For ease of comparison, the leasing cost in each day is normalized by the corresponding cost of the LB-C approach. Our CACL-based approach has the lowest cost, decreasing 16.9%-19.5% of LB-C approach and 17.8%-20.4% of LB-V approach. Another observation is that the leasing cost on Feb03 is higher than those of the other two days in all approaches, because the number of broadcasters on Feb03 is the highest. We also plot the normalized leasing cost and the average percentage of migration from dedicated servers to public clouds during our whole datasets in Figure 3.17b, we can observe that the decreasing cost shows the weekly pattern and our approach provides the elastic workload provisioning cost-effectively. Moreover, more than 30% of broadcasters are migrated to public clouds in every day. Our simulation results show that compared with hosting all broadcasters in dedicated servers, leasing flexible instances on public clouds to migrate the workload of certain broadcasters is a cost-effective solution.

3.6 Summary

In this chapter, we examined the crowdsourced livecast platforms, which provide live streaming service and live chatting service to Internet users. The results from Twitch-based measurement indicated the potential issues therein. In particular, a large number of unpopular broadcasters consume the valuable dedicated resources continuously. Through Amazon EC2-based measurement, we analyzed the feasibility of migrating a part of these broadcasters to public clouds. To accommodate unpredictable workloads and realize the adaptive offloading in demand, we proposed the Cloud-assisted Crowdsourced Livecast (CACL) for the initial offloading, as well as the ingesting redirection and transcoding assignment. Our trace-driven simulations demonstrated the cost-effectiveness of the CACL framework.

Chapter 4

Exploring Viewer Gazing Patterns for Touch-based Mobile Gamecasting

Empowered by today's high-performance mobile devices and communication networks, we have witnessed an explosion of *mobile gamecasting (MGC)* as a must-have application on gamers' mobile devices. Such MGC platforms as YouTube Gaming, Twitch, and Mobcrush, have ushered in a new wave of innovations in how multimedia content is created and consumed, distributing a game playthrough from a gamer's (i.e., broadcaster's) personal device to a large population of viewers. The recent news from Twitch reveals that more than 35% of the viewership are mobile viewers in all its gamecasting channels every month. A research report from Google also indicates that a third of the U.S. mobile gamers are defined as "avid gamers", who spend more than nine hours a week on average playing mobile games on smartphones; moreover, the more time these "avid gamers" spend on YouTube, the more time they spend on gaming. These MGC applications, as the propellents in both streaming and gaming markets, are posing significant challenges to the existing live broadcasting platforms, particularly with mobile broadcasters and viewers.

Several studies [72, 59] have already investigated the opportunities on the broadcaster-side. Yet the oscillation of source quality may affect the viewers' QoE (Quality-of-Experience) greatly. Recently, the rapid development of eye-tracking research makes foveated-aware optimization of viewers' watching experience possible, which has seen use cases for content distribution and live streaming [60] [2]. They are not targeting MGC applications, and typically need eye-tracking peripherals to collect the viewers' gazing data in real-time, which in turn produces extra energy consumption on mobile devices.

To explore the opportunities in the MGC context, we measure the data traces collected from gamers and viewers. The results show that strong correlations exist between the gamers' interactions on touch screens and the viewers' gazing patterns. Motivated by this observation, we propose a novel interaction-aware optimization framework that guides

Table 4.1: Classification and definition of gamer's touch interactions

Touch Interaction	Definition	Examples
Single-touch (ST)	press once and release quickly	press a button or activate an object in a game
Press-drag (PD)	first press a few seconds to activate/select a game element and then drag it to a target	move a card onto a target area or deploy attacking path using soldiers
Pan (PA)	omnidirectional one-finger swipe in mobile games	expand the field of view when game scene is larger than screen size
Zoom (ZM)	a double-touch interaction, two fingers are used to scale up/down the game view	display (or hide) details before attacking enemy camps

mobile gamecasting in advance, even before the source encoding step. The target and key challenges towards designing the framework lie in three aspects: (1) We need to understand the characteristics of the gamers' interactions and the viewers' gazing patterns with dynamic game strategies and eye movements. (2) We need online prediction to find the correlations with no the assistance from eye-gazing peripherals preferably. (3) We need to design an optimization strategy to improve the energy efficiency and adjust the stream quality using the predicted gazing patterns.

To address the above problems, we first classify the users' behaviors into distinct groups, including single-touch, press-drag, pan, and zoom for touch interactions (as shown in Table 4.1), and area-fixation, smooth-pursuit, and scene-saccade for gazing patterns (as shown in Table 4.2), corresponding to the steady, slow, and fast movements of human eyes. Our framework then incorporates a touch-assisted prediction (TAP) module and a tile-based optimization (TBO) module. The former achieves offline training and online prediction by building association rules [73], and the latter improves the energy efficiency in mobile devices using a tile-based quality selection with bandwidth and QoE constraints. Trace-based simulations and a user study demonstrate that our framework achieves noticeably better QoE under similar network constraints.

4.1 Background

Recent years have witnessed an explosion of gamecasting applications, in which gamers broadcast their game playthroughs in real-time [79]. Such pioneer platforms as YouTube Gaming, Twitch, and Mobcrush have attracted a massive number of online broadcasters, and each of them can have hundreds or thousands of fellow viewers. As shown in Figure 4.1, a typical MGC platform maintains two services: (1) *a live streaming service*, which not only fulfills the encoding, ingesting, transcoding, and distribution of live streams, but also implements the screen recording functionality on mobile devices; and (2) *a live chat service*,

Table 4.2: Classification and definition of viewer's gazing patterns

Gazing Pattern	Definition	Examples
Area-fixation (AF)	gaze on a fixed area	spend more time to watch the center of gamecasting if there is no player's touch interaction
Smooth-pursuit (SP)	gaze smoothly follows the movement of an object	focus on the moving game cards or buildings when gamers change strategies or deployments
Scene-saccade (SS)	move eyes between two or more fixation areas quickly	read a notice board or item descriptions

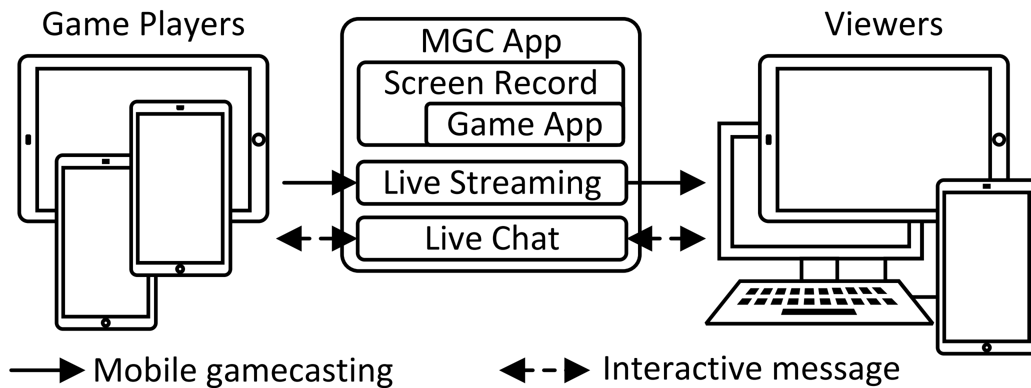
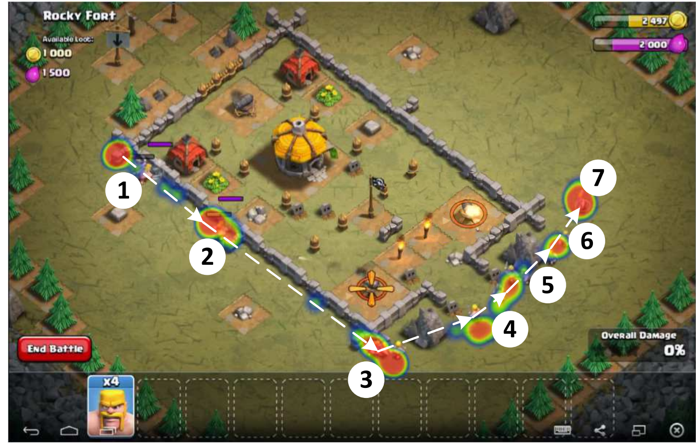
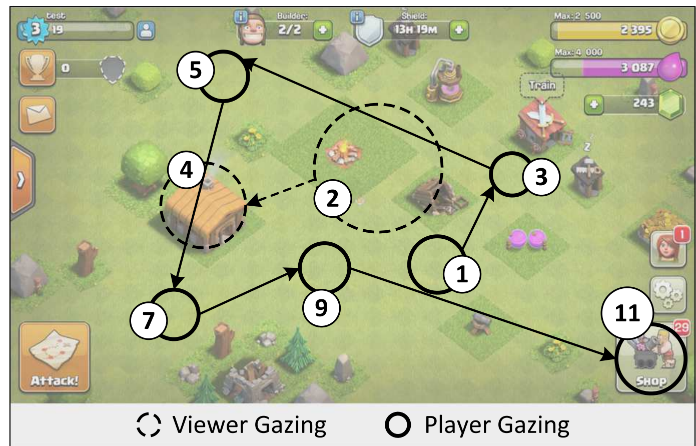


Figure 4.1: A generic architecture of the MGC platform



(a) Example A

(b) Example B



(c) Gazing pattern comparison

Figure 4.2: Motivations of our study

which exchanges the users' messages through IRC (Internet Relay Chat) or proprietary chatting protocols. Some recent studies have already focused on gamecasting platforms by

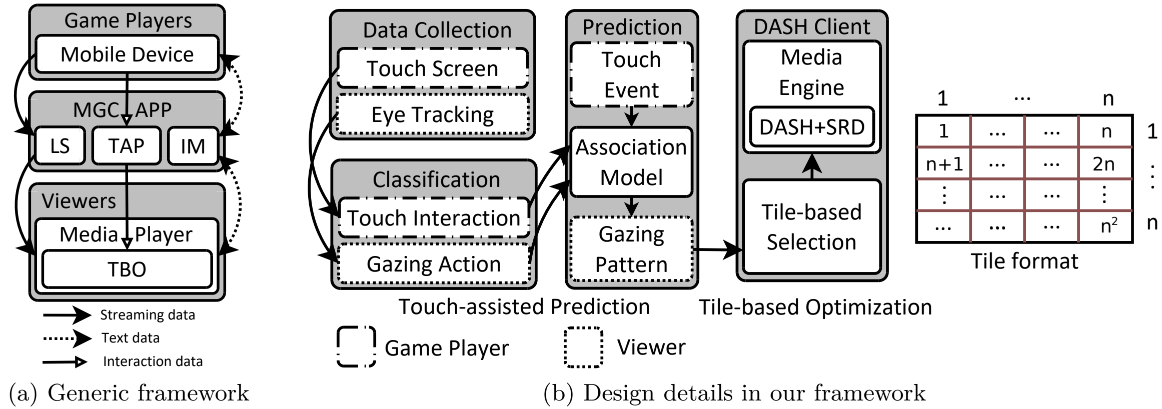
proposing novel frameworks [26] or optimizing the live streaming service [6]. Compared with these studies, we focus on a novel branch of gamecasting services: mobile gamecasting. We therefore investigate its unique feature, i.e., gamers' touch interactions, to optimize the gamecasting transmission.

A representative MGC scenario is illustrated as follows: a gamer first launches a mobile game using an MGC application, e.g., YouTube Gaming App, on a smartphone. The top bar on the screen provides the controllers for mobile cameras, microphone, screen recording, and other configurations. After clicking the screen recording button, the gamer can use this MGC application to encode the recorded game scenes as a live stream and transmit it to an ingesting server. After multi-version transcoding, the segments of this live stream are delivered to a large number of heterogeneous viewers. During the gamecasting, the gamer and the viewers can closely interact by lively discussing game strategies via a chat service. Yet the high-performance mobile devices suffer from energy constraints with the built-in batteries, but also enjoy opportunities with novel operation interfaces, in particular, the touch screens. Several works have been devoted to analyzing the touch behaviors for specific applications, e.g., recognizing users [12]. Zhang *et al.* [82] examined the instant video clip scheduling problem based on the unique scrolling behaviors on mobile devices. Our touch-assisted prediction is motivated by these works, but we focus on the gamers' touch interactions to predict the viewers' gazing patterns for MGC applications.

Our tile-based optimization is motivated by the works in [66, 47]. They mainly employ saliency models to predict Region-of-Interest (ROI) in live streams, but can hardly capture the patterns of human gaze in real-time. To overcome these challenges, recent works have designed content transmission based on the viewers' gazing information in live streaming services [60] and cloud gaming systems [2]. These works need extra support from eye-tracking devices, e.g., a web-camera, to capture the gazing data during the whole process. Our work differs from them in that we predict the viewers' gazing patterns based on the gamers' interactions in MGC applications.

4.2 Motivation

In this section, we optimize MGC applications from a new perspective. That is, through analyzing the gamers' touch interactions on touch screens, we predict the viewers' gazing patterns towards energy-efficient streaming. We first answer the following question: *How do a gamer's interactions affect the viewer's gazing patterns (including the focusing regions and movements)?* To investigate the associations between them, we capture the gamers' touch data and the viewers' gazing data through our testbed, which consists of a smartphone, an eye-tracking device, and a desktop PC. We connect the smartphone to the desktop PC to record the gamers' touch data and deploy the eye-tracking device to capture the viewers'



(a) Generic framework (b) Design details in our framework

Figure 4.3: Interaction-aware optimization framework in MGC

Event Type

Timestamp Multi-Touch Event Value

[2213.341595]	EV_ABS	ABS_MT_TRACKING_ID	00000024
[2213.341638]	EV_ABS	ABS_MT_POSITION_X	00000201
[2213.341655]	EV_ABS	ABS_MT_POSITION_Y	0000040b
[2213.341671]	EV_ABS	ABS_MT_TOUCH_MAJOR	00000008
[2213.341683]	EV_ABS	ABS_MT_TOUCH_MINOR	00000007
[2213.341750]	EV_SYN	SYN_REPORT	00000000
[2213.415273]	EV_ABS	ABS_MT_TRACKING_ID	ffffffff

Touch Event Touch Interaction

Figure 4.4: Sample of touch screen events

gazing data. The details about the testbed configurations will be introduced in Sections 4.4 and 4.5, respectively. Here, we first highlight our findings.

Our testbed experiments show that the gamers' touch interactions change the mobile game scenes and objects, which in turn pilots the viewers' gazing patterns. Figure 4.2 gives two examples to illustrate their associations using the viewers' gazing heatmap. In Figure 4.2a, a gamer designs an attacking path to deploy soldiers from region #1 to #7. Consequently, a viewer's gazing points also follow this path. This example implies that the gazing regions correspond to the touch interaction regions but have a temporal delay. In Figure 4.2b, a gamer touches the button in region #1 and activates the system setting options. Afterward, a viewer focuses on reading the information on each button. This example shows that the gazing regions do not always have spatial correlations with the touch regions, but their associations still can be found by analyzing the gamers' touch interactions and the viewers' gazing patterns together.

Yet there is a new question: if a gamer considers the game strategies through investigating a game scene first, and then decides a touch interaction in the next step, *why cannot*

Table 4.3: Statistics of touch data

Game ID	Game Name	# of Events	# of Interactions	Game Genre
G1	Clash of Clans	65940	684	Multiplayer online strategy game
G2	HearthStone	33422	250	Collectible card game
G3	Clash Royale	37151	421	Multiplayer online battle arena, collectible card games, tower defense

we directly rely on the gamer's gazing data for the viewer's gazing prediction? Such a strategy, however, needs to capture the gamer's eye movements in real-time, which needs either plugin supplements (e.g., a mobile camera) with higher energy consumption or expensive eye-tracking glasses. There are also discrepancies among different gamer and viewer groups. To illustrate it, we use an example[1] to compare the gazing differences between a gamer and a viewer in Figure 4.2c. In this figure, we plot the circle regions to exhibit the gazing movements of a gamer and a viewer, respectively. We can find that the gamer proactively focuses on lots of areas to determine the next action (i.e., click a button and open a shopping list). The gazing sequence is: the empty fields (regions #1, #3, #5, #7 and #9), and then the SHOP button (region #11). On the other hand, the viewer pays much attention to the center area in region #2, and then focuses on the Town Hall in region #4. Since a gamer must observe the game objects carefully to determine the next game strategy, s/he has more complex and unpredictable gazing patterns compared with a viewer. From these examples, we also see an association sequence between the gamers' and viewers' behaviors: gamers' gazing behaviors $\rightarrow$ gamers' game strategies $\rightarrow$ gamers' touch interactions $\rightarrow$ viewers' gazing patterns. A gamer first observes the game scenes and thinks about the game strategies; different strategies then generate the corresponding touch interactions. When a viewer watches this game video, the gazing patterns are based on two factors. First, the human eyes prefer to gaze on the center area of a scene; second, when a gamer touches or activates any object, the video scene will be changed significantly, attracting the viewer's attention. Two video examples[2] show the relationships among the touch interactions and gazing patterns of a gamer and the gazing patterns of a viewer.

4.3 Interaction-Aware Design

Motivated by these observations, we design an interaction-aware optimization framework for MGC platforms, as shown in Figure 4.3a. Our design incorporates two new modules:

[1]Example link: `https://youtu.be/EP2v9m9d15E`

[2]1. `https://goo.gl/2WsdtP`, 2. `https://goo.gl/XNS8Bu`

Touch-Assisted Prediction (TAP) and Tile-Based Optimization (TBO). The TAP module predicts the viewer's gazing patterns through ingesting the gamers' touch interactions and relays the prediction results to the TBO module, which then accordingly optimizes the energy and bandwidth consumption.

4.3.1 Touch-assisted Prediction Module

As shown on the left part in Figure 4.3b, the TAP module involves three steps: Data Collection, Data Classification, and Gazing Prediction. We highlight their design concepts here and present the details of each step in the following sections.

- Data Collection: We recruit a set of gamers and viewers for training. We first collect these gamers' touch events and record the game videos when they play a mobile game. We then capture these viewers' gazing points when they watch the selected videos. These training data will be formatted and processed towards the next step.
- Data Classification: According to game-specific rules, we classify the touch events and gazing points into pre-defined groups. The gamers' touch interactions include single-touch, press-drag, pan, and zoom. The viewers' gazing patterns consist of area-fixation, scene-saccade, and smooth-pursuit.
- Gazing Prediction: The main part of this step is to build an association model, which is derived from association rules learning. The prediction module receives the gamers' touch interactions and obtains the predicted viewers' gazing patterns during mobile gamecasting.

4.3.2 Tile-based Optimization Module

In the TBO module, as shown on the right part in Figure 4.3b, our framework is based on the Spatial Relationship Description (SRD) feature in the recent MPEG-DASH (Dynamic Adaptive Streaming over HTTP) amendment [55]. SRD works to stream a subset of spatial sub-parts of a video to viewers' devices. Every frame of a short content in a video is first partitioned into multiple frames of smaller resolution. Then, these neighboring smaller frames in the same region are combined into an HTTP-based tiled content (*tile* in short). Finally, a viewer's media player renders a sequence of tiles to reconstruct this video. Every tile, therefore, contains a part of the video during a short interval. A tile-based optimization algorithm is then designed to adjust the quality of every tile according to the predicted viewers' gazing patterns. We partition every short stream into $n - by - n$ tiles following the works in [55, 42]. In practical scenarios, the tile size can be adjusted to meet different requirements [42], for example, 4x2 in HD videos [55]. In our study, the default n is set to be 5.

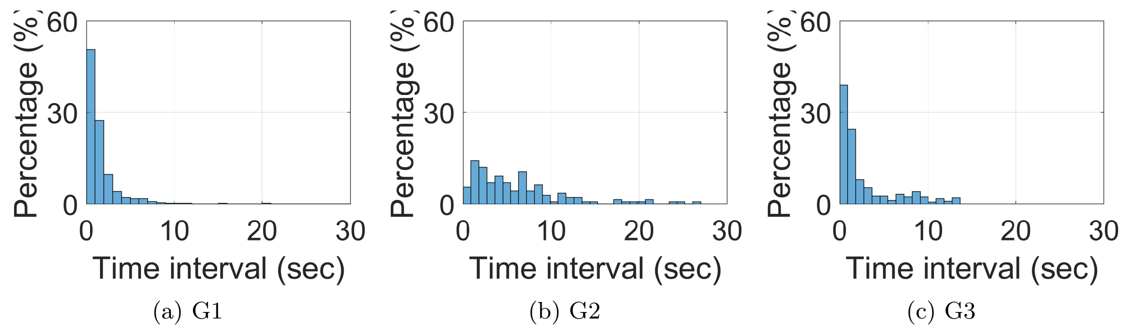
4.4 Understanding Game Touch Interactions

In this section, we investigate the gamers' touch interactions based on real-world data traces and classify game-specific touch interactions.

4.4.1 Touch Data Collection

To collect the gamers' touch data, we install the Android Debug Bridge (ADB)[3] on a desktop PC (DELL Optiplex 7010) that connects to a mobile phone (Samsung Galaxy S5 with Android 6.0.1).

Figure 4.4 shows a sample of touch screen events, where each line is an event with four fields: timestamp, event type, multi-touch event, and value. According to the multi-touch events and the event values, we can distinguish different touch interactions. This sample represents a single-touch interaction, which means a gamer quickly touches the screen center once. The dynamics of *touch position* (ABS_MT_POSITION_X/Y) and *touch area* (ABS_MT_TOUCH_MAJOR/MINOR) are recorded in a trace file. We write a Python script to extract the event properties of every touch interaction from the original data traces. The formatted touch interaction is a 5-tuple: {ID, start_timestamp, position_array, area_dynamics, duration}.



(a) G1 (b) G2 (c) G3

Figure 4.5: Time intervals between consecutive touch interactions

To understand the characteristics of distinct gamers, we recruited ten volunteers[4]. Each of them individually plays a game on S5 for two minutes. To explore the impact of game genre, we select three popular games: G1-Clash of Clans, G2-HearthStone, G3-Clash Royale and capture the screens during game playing. The selected three games not only attract a huge number of audiences in various gamecasting platforms, but also achieve high revenue

[3]ADB is a versatile command line tool that allows users communicate with connected Android devices.

[4]Gender, female/male: 2/8; Age, (20-25)/(26-30)/(>30): 3/5/2; Game experience, expert/beginner: 7/3, https://eyegazing.github.io/

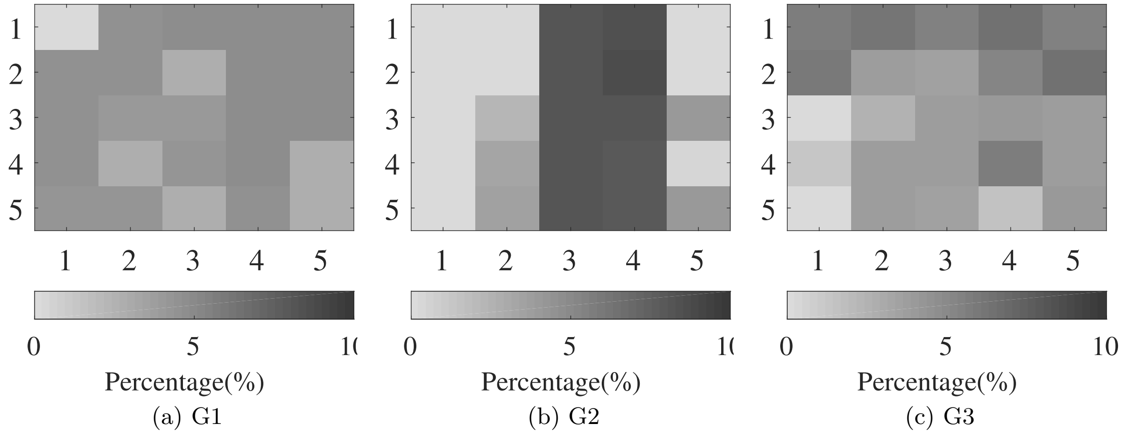


Figure 4.6: Characteristics of touch regions

in the mobile gaming market over the world[5]. Table 4.3 presents the details of our data traces. We find that the number of touch interactions is mostly determined by the gamers' preferences. There is no strict proportion between the number of events and the number of interactions. We also investigate the time interval between two consecutive touch interactions. As shown in Figure 4.5, the time intervals in G2 are higher than those in G1 and G3.

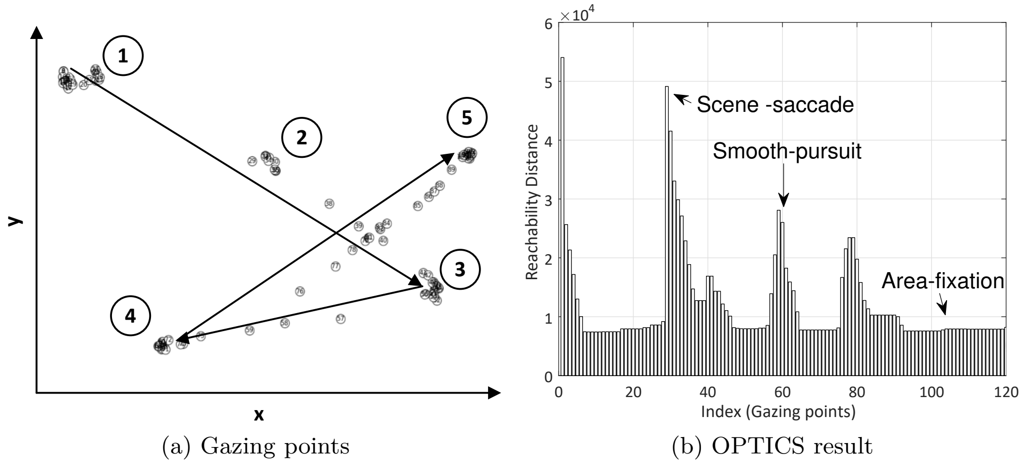
4.4.2 Interaction Classification

According to the official description of touch actions in the Android Design Documentation[6], we define the following four game-specific interactions: *Single-Touch* (ST), *Press-Drag (PD)*, *Pan* (PA), and *Zoom* (ZM), as shown in Table 4.1. These touch interactions are frequently used in mobile games. We use a decision tree [73] to classify them with five key features extracted from the data: duration, position, direction, maximum touch area, and minimum touch area. Corresponding to the touch interactions, this decision tree has four outputs: single-touch, press-drag, pan, and zoom. To train the decision tree and test the accuracy, we use 300 touch interactions labeled by the gamers. The decision tree achieves an average accuracy of 97% for the classification of the four touch interactions, which is adequate for the association learning in our framework.

As shown in Figure 4.6, we divide the touch screen into 25 regions, the vertical color-bar exhibits the mapping of the percentages into the color-map on the left part. We can find that (1) G1 gamers touch almost all the areas, except for area #1, because the gamers have to frequently carry out strategies in G1; (2) G2 gamers seldom touch the left-side regions,

[5]`https://www.superdataresearch.com/market-data/`

[6]`https://material.google.com/patterns/gestures.html`



(a) Gazing points (b) OPTICS result

Figure 4.7: Example of gazing points classification

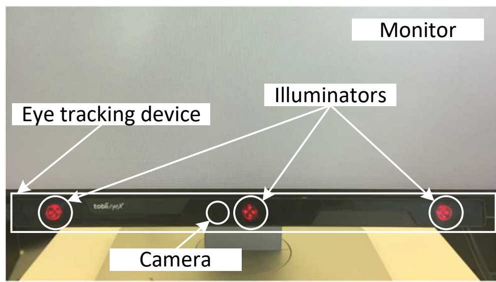


Figure 4.8: Eye tracking device in our test-bed

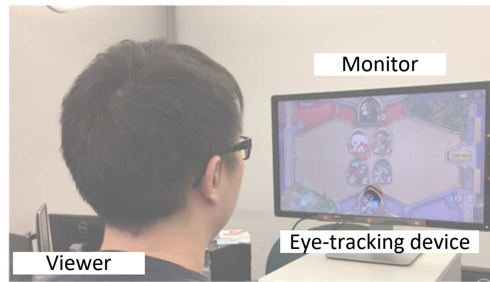


Figure 4.9: An illustration of collecting a viewer's gazing data

because most of the objects controlled by the gamers are located on the right-side and the bottom-side of the touch screen; (3) G3 gamers prefer the top-side and right-side areas[7], which is also determined by the game design.

4.5 Insights into Viewers' Gazing Patterns

In this section, we first propose the data collection method in the viewers' gazing investigation. Then, we analyze the characteristics of the viewers' gazing data.

4.5.1 Gazing Data Collection

We choose Tobii eyeX[8] as the eye-tracking device to collect the viewers' gazing data due to its affordable price, suitable sampling rate, and high accuracy. It is connected to a desktop

[7]Because G3 is a portrait-oriented game, the top-side and right-side in Figure 4.6c are the right-side and bottom-side of the portrait-oriented touch screen, respectively.

[8]The refreshing rate: $> 60Hz$; the operating range: 50-90cm.

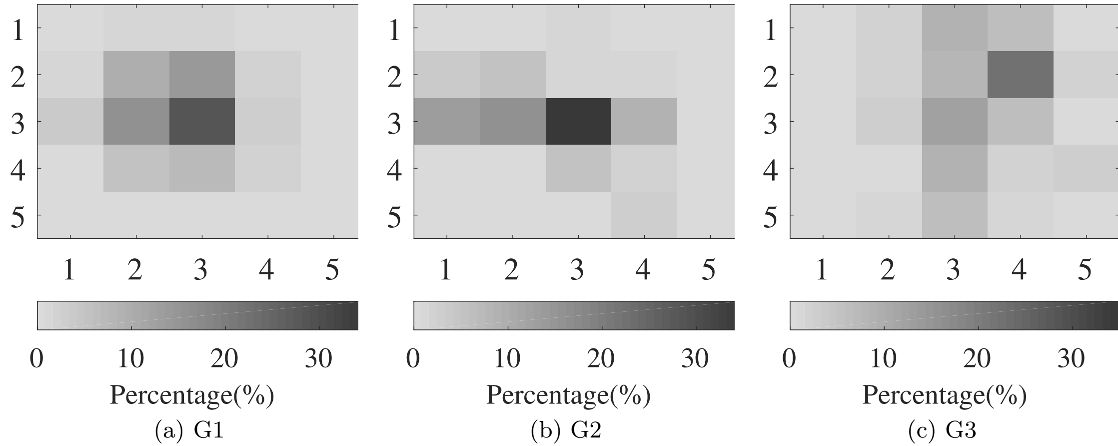


Figure 4.10: Characteristics of gazing regions

PC (DELL Optiplex 7010) through a USB 3.0 port and attached to the frame of a 27-inch monitor (DELL U2715H), as shown in Figure 4.8. The eye-tracking device consists of three illuminators and one camera. Figure 4.9 illustrates that a viewer's gazing data is collected by the eye-tracking device. When the viewer watch a video, the illuminators create patterns of near-infrared light on his/her eyes. Then, the camera captures high-resolution images of the eyes. Finally, the built-in algorithms analyze these images and calculate the corresponding gazing coordinates on the monitor. The ten volunteers mentioned earlier have also assisted us to collect gazing data. Each of them have personal profiles to calibrate the eye-tracking device before the data collection. As a viewer, every volunteer watches three two-minute game videos selected from a gamer in Section 4.4.

4.5.2 Gazing Classification

We define the following three gazing patterns: *Area-Fixation* (AF), *Smooth-Pursuit* (SP), and *Scene-Saccade* (SS), which are based on state-of-the-art eye-tracking research [20], as shown in Table 4.2. Figure 4.7a shows several examples of these patterns. In this figure, we use the labeled circles to indicate the AF patterns of a viewer. The viewer's attention is quickly changed from region #1 to region #2, which corresponds to an SS pattern. Then, we observe three SP patterns among areas #2, #3, #4, and #5.

After investigating the gazing points, we find that perfectly classifying gazing points into the three patterns is impossible due to data noises. As such, we first pre-process the gazing points to improve the accuracy. As shown in Figure 4.7a, if a viewer gazes on a fixed area, the gazing points are clustered in a two-dimensional space. Moreover, every gazing point has a temporal dimension, i.e., its timestamp, so we employ the OPTICS (Ordering Points To Identify the Clustering Structure) algorithm [5] to pre-process these gazing points. The OPTICS algorithm finds the density-based clusters (i.e., the AF patterns in our data traces)

through calculating the distance between two gazing points. This distance also reflects the speed of the movement of human eyes from one area to another. Figure 4.7a shows a pre-processing example, which depicts five AF patterns, one SS pattern (from area #1 to #2), and three SP patterns (#2→#3, #3→#4, and #4→#5). Figure 4.7b shows the corresponding pre-processing results using the OPTICS algorithm. According to the results, we extract the features of the viewers' gazing patterns, including the time interval and the reachability distance[9]. Similar to the classification of the touch interactions, 100 labeled patterns are used to train the decision tree, which achieves an average accuracy of 96% for the classification of the three gazing patterns.

To further investigate the viewers' gazing patterns, we use a similar approach as in Section 4.4.2 to examine the characteristics of the gazing regions. According to the setting of divided tiles, we define "gazing region" to illustrate which tile is gazed by a viewer. Figure 4.10 plots the percentages of the gazing regions in the three games. Note that the viewers exhibit distinct gazing preferences in different games: (1) G1 viewers mostly gaze on the left part of gamecasting; (2) G2 viewers focus on the middle part; (3) G3 viewers prefer the top part. This implies that strong correlations exist between the gamers' touch interactions and the viewers' gazing patterns in the MGC context.

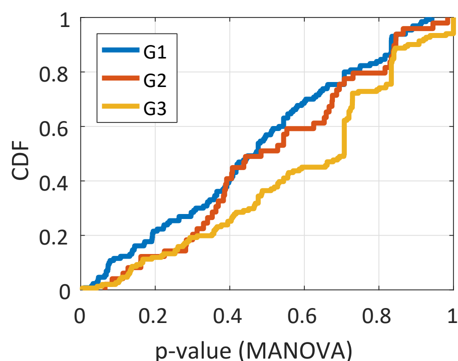


Figure 4.11: CDF of p-value

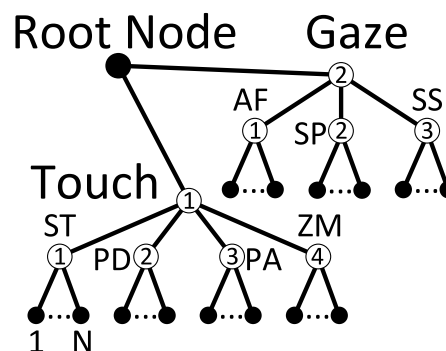


Figure 4.12: Encoding tree

To further examine the associations between the touch interactions and the gazing patterns, we choose Multivariate ANalysis Of VAriance (MANOVA) [8] to statistically analyze different viewers' gazing regions between the start of a touch interaction and the start of the next one. In MANOVA, a high p-value means a high similarity of the gazing regions among different viewers. Figure 4.11 shows the Cumulative Distribution Function (CDF) of the p-values in the three games. The mean for each game is higher than 0.45. Besides, more than 60% of results are higher than 0.4, which implies that the viewers' gazing patterns are similar after the same touch interactions.

[9]The reachability distance from point g_1 to g_2 is equal to the maximum value two types of distances: (1) the distance from point g_1 to g_2; and (2) the distance from point g_1 to the core point in its cluster.

Table 4.4: Encoding example

Data traces	Encoded transactions
{ST, (365, 632)};{AF, (250,430),(255,439)}	T_1. {1113, 217}
{PD, (375, 700), (300, 1000)}; {SP, (280,500),(255,900)}	T_2. {1213, 1218, 227, 2217}

4.6 Touch-Gaze Association Learning

Association rules describe strong relations of the items that occur frequently together in a data set. In this section, we first introduce the preliminaries of association rule learning, and then build up such associations in the MGC scenario.

4.6.1 Preliminaries

The inputs of association rules learning contain: (1) *items data*, A_i, where item $A_i \in I, i = \{1, \cdots, M\}$, and (2) a *transaction set*, T, which consists of a set of transactions $< T_i, \{A_p, \ldots, A_q\} >$, where T_i is a transaction identifier and $A_i \in I, i = \{p, \cdots, q\}$. A collection of zero or more items is defined as an itemset. If an itemset contains k items, it is called k-itemset. An association rule is defined as an implication expression of form $X \rightarrow Y$, where $X \bigcap Y = \emptyset$ and $X, Y \subseteq I$. The *item support count* $\delta(X)$ of itemset X gives the number of the transactions that contain a particular itemset. $\delta(X) = |\{T_i | X \in T_i, T_i \in T\}|$. To find the frequently occurred rules, *support* and *confidence* also need to be defined. Support determines how often a rule is applicable to a given data set, while confidence determines how frequent items in Y appear in transactions that contain X. Support $s(X \rightarrow Y)$ is defined as follows: $\delta(X \bigcup Y)/M$. Confidence of a rule $X \rightarrow Y$ is accordingly defined as $c(X \rightarrow Y) = \delta(X \bigcup Y)/\delta(X)$.

To discover frequent itemsets and build reasonably strong associations, we must specify two thresholds: *minimum support*, s', and *minimum confidence*, c'. The first step is to find all the itemsets that satisfy threshold s'. These itemsets are called *frequent itemsets*. The second step is to extract all the rules from the frequent itemsets found in the previous step such that the confidence of these rules is no less than threshold c'. The second step is straightforward, while the first step needs more attention since it involves searching all possible itemsets. The classical Apriori algorithm [73] employs a bottom-up approach by identifying the frequent individual items in the transaction data set and extending them to larger itemsets while the items satisfy the minimum support threshold. The frequent itemsets returned by the Apriori algorithm are then used to determine the association rules.

4.6.2 Touch-Gaze Association

To map our data into the corresponding transactions T_i, we treat the touch interactions and the gazing patterns as items and each transaction in our study as a combination of several

touch items and gazing items from the start time of one touch to the start time of next one. To simplify the discussion and fit the tile-based optimization, we partition the touch items and the gazing items into N groups according to their positions, where $N = n^2, n \in \mathbb{Z}^+$. Each item is then encoded based on the tree structure in Figure 4.12, where the fourth layer represents the order of groups. Thus, given a touch interaction or a gazing pattern, we can encode it based on its classification and position. Table 4.4 shows one encoding example. In this example, we set $N = 25$. $\{ST, (365, 632)\}$ and $\{AF, (250, 430), (255, 439)\}$ mean that (1) a gamer touches $(365, 632)$ once before the next interaction; and (2) a viewer's gazing points contain $(250, 430)$ and $(255, 439)$, which is an area-fixation pattern. According to the setting of tiles, two gazing points are in a region, thus they are encoded in an item $\{217\}$. The encoded transaction T_1 includes two items $\{1113\}$ (Touch-ST-13) and $\{217\}$ (Gaze-AF-7). The rationale of this encoding method is that each encoded item contains all key information we need.

Algorithm 3: Touch-Gaze Association Learning

Input:
(1) A: the list of encoded touch interactions;
(2) B: the list of encoded gazing patterns
Output: G, Association rules

```
1  create an empty transaction set T
2  for i from 1 to |A| do
3      create an empty transaction T_i
4      add touch interaction A_i in T_i
5      if A_i is not the last touch interaction then
6          add the gazing patterns that occur between A_i and A_{i+1} in T_i
7      else
8          add the gazing patterns that occur after A_i in T_i
9      add T_i in T
10 generate frequent itemsets G from T using the Apriori Algorithm
11 for each itemset g in G do
12     if all items in g belong to a type of behavior then
13         remove g from G
14 return G;
```

We summarize the touch-gaze association learning in Algorithm 3. Because lots of gazing patterns may occur after a touch interaction, the algorithm first adds the encoded gazing items into a sequence of transactions according to the time intervals between consecutive touch items(lines 2 to 9). Then, we use the Apriori algorithm to find all frequent itemsets for mining strong association rules in the encoded transactions (line 10). Since we aim to find the association rules between the touch items and the gazing items, we remove the frequent itemsets that only contain touch items or gazing items (lines 11 to 13). For example, we

cannot find association rules from frequent itemset $\{211, 212, 235\}$, because all items in it are gazing items.

Given the frequent items, we can acquire the association rules to predict the viewers' gazing patterns. Through employing association rules for new touch data, the TAP module will predict the gazing patterns before the next touch interaction. As mentioned, the gazing region (i.e., the tile gazed by viewers) is related to the viewers' QoE. We thus define four gazing pattern sets to represent the importance of gazing regions: (1) if an AF pattern exists in a gazing region, we add the order of this region into set L_1; (2) similarly, let L_2 denote the set of the regions that contain SP patterns; (3) L_3 is the set of the regions that include SS patterns; (4) L_4 is the set of other regions that do not have gazing patterns. Finally, the result L of the predicted gazing patterns is sent to the TBO module, where $L = \{L_1, L_2, L_3, L_4\}$.

4.7 Tile-based Optimization

In this section, we first model the tile-based optimization problem with bandwidth and QoE constraints and transform it into an equivalent problem with an efficient solution.

4.7.1 Problem Formulation

For ease of exposition, we assume that the duration D of a short stream varies from a few seconds to several minutes (e.g., two seconds in our experiments). Every short stream is reconstructed by N tiles and each tile has V quality versions. We thus denote each tile as $t_{i,j}$, $i \in \{1, \cdots, N\}, j \in \{1, \cdots, V\}$. Let $d_{i,j}$ and $e_{i,j}$ be the size and downloading energy consumption of tile $t_{i,j}$, respectively. We denote the quality of a tile by $q_{i,j}$, which is a concave increasing function of the encoding bitrate. The size and encoding bitrate of a tile can be acquired from the streaming servers, and the downloading energy consumption can be estimated [32]. We use S to denote the duration of the buffered stream on the viewer-side. The tile-based optimization can thus be formulated as to maximize the tile quality per energy consumption.

$$Maximize : \sum_{i=1}^{N} \sum_{j=1}^{V} \frac{q_{i,j}}{e_{i,j}} x_{i,j} \tag{4.1}$$

subject to:
Streaming Availability Constraints (4.2) and (4.3)

$$\frac{\sum_{i=1}^{N} \sum_{j=1}^{V} d_{i,j} x_{i,j}}{B} \leq S \tag{4.2}$$

$$\sum_{j=1}^{V} x_{i,j} = 1, i \in \{1, \cdots, N\}, x_{i,j} = \{0, 1\} \tag{4.3}$$

Foveated Quality Constraints (4.4), (4.5) and (4.6)

$$v(t_{i,a}) \geq \alpha, t_{i,a} \in L_1 \tag{4.4}$$

$$0 \leq v(t_{i,a}) - v(t_{i,b}) \leq \beta, t_{i,a} \in L_k, t_{i,b} \in L_{k+1} \tag{4.5}$$

$$v(t_{i,a}) - v(t_{i,b}) = 0, t_{i,a}, t_{i,b} \in L_k, k \in \{1, \cdots, 4\} \tag{4.6}$$

where B is the average bandwidth, which is estimated according to the current gamecasting session on the viewer-side, and α and β are two tunable parameters. The rationale of the Streaming Availability Constraints is as follows: (1) all tiles should be downloaded completely before the buffer becomes empty, which guarantees smooth playback of a gamecasting; (2) the clients only need to download one quality version for every tile, which avoids extra bandwidth consumption. The rationale of the Foveated Quality Constraints is as follows: (1) to improve the quality of the tiles in gazing pattern set L_1, we denote the minimum quality version of these tiles by a parameter α; (2) we can select the version of the tiles in different gazing pattern sets to meet the streaming availability constraints and achieve the optimization target, but the version gap between two neighboring pattern sets cannot be larger than β considering that a large value will impact viewers' QoE; (3) the tiles in a gazing pattern set should have the same version.

4.7.2 Solution

If we only consider the Streaming Availability Constraints, the tile-based optimization problem can be transformed into a Multiple Choice Knapsack (MCK) problem, which is NP-hard with practically efficient solutions available (e.g., pseudopolynomial-time dynamic programming) [39]. It is worth noting that the optimal solution of this MCK problem may not meet the Foveated Quality Constraints. We therefore propose Algorithm 4. The inputs include (1) parameters α and β; (2) the prediction result L; and (3) the version of tiles, and the corresponding size and estimated energy consumption. The algorithm first narrows down the solution space (lines 2 to 6) according to the Foveated Quality Constraints before employing dynamic programming. If there exists a feasible solution (line 7), the algorithm solves the MCK problem using a dynamic programming approach; otherwise, it uses a parameter *index* to alternately adjust β and α to enlarge the solution space (lines 10 to 16). Finally, we can obtain the version selection of every tile (line 18).

Algorithm 4: Tile-based Selection Optimization

Input:
(1) T_{info}, tile information, including the version info., size info., and energy info. for all tiles;
(2) L, the result of the predicted gazing patterns;
(3) α, the minimum version for the tiles in L_1;
(4) β, the maximum version gap between the tiles in L_k and L_{k+1}.
Output: Q, the version selection for all tiles

```
1  index = 0
2  for each gazing pattern set l in L do
3      for each tile t in l do
4          for each quality version v of t do
5              remove v if it does not meet the Foveated Quality Constraints
6          update T_info
7  if the time of downloading the lowest versions of all tiles is less than the duration
     of buffered stream then
8      acquire optimal version selection Q using a dynamic programming solution of
         the MCK problem
9  else
10     //update β and α alternately to enlarge the solution space
11     if index mod 2 == 0 then
12         β ← β + 1
13     else
14         α ← α − 1
15         β ← β − 1
16     index ← index + 1
17     goto line 2
18 return Q;
```

(a) G1 (b) G2 (c) G3

Figure 4.13: Eye gazing patterns vs. Prediction results

4.8 Performance Evaluation

In this section, we examine the performance of the proposed framework by the following three steps: we first evaluate the performance of the touch-assisted prediction module based on three state-of-the-art similarity metrics; then, we collect real traces to examine the performance of the tile-based optimization; finally, we conduct a user study to compare the viewers' QoE under a variety of configurations.

4.8.1 Performance of Touch-assisted Prediction

In computer vision research, saliency models generate saliency maps to predict where humans look at images. Similarly, our touch-assisted prediction module can be used to generate saliency maps to reflect the gazing areas of viewers. To examine the performance of this module, we choose three state-of-the-art metrics [9]: Normalized Scanpath Saliency (NSS), Area Under the Curve (Borji implementation, *AUC-Borji* in short), and Area Under the Curve (Judd implementation, *AUC-Judd* in short). We use the three metrics to compare the similarity between the predicted saliency maps and the ground truth collected by our eye-tracking device. Higher NSS, AUC-Borji, and AUC-Judd values indicate high-valued predictions of viewers' gazing areas. The theoretical ranges of NSS, AUC-Borji, and AUC-Judd are $[-\infty, \boldsymbol{\infty}]$, $[0, \mathbf{1}]$, and $[0, \mathbf{1}]$, respectively (best score in bold). Based on the setting of tiles in this paper, every predicted saliency map consists of white/grey/black tiles, as shown in Figure 4.13. In these examples, the white tiles show that these areas include AF patterns, the grey ones contain SP and SS patterns, and the black ones do not have any gazing patterns. We further plot the percentage of all metrics to show the prediction performance in Figures 4.14, 4.15, and 4.16. From these figures, we observe that the touch-assisted prediction achieves good performance for the three games with all the three metrics. The

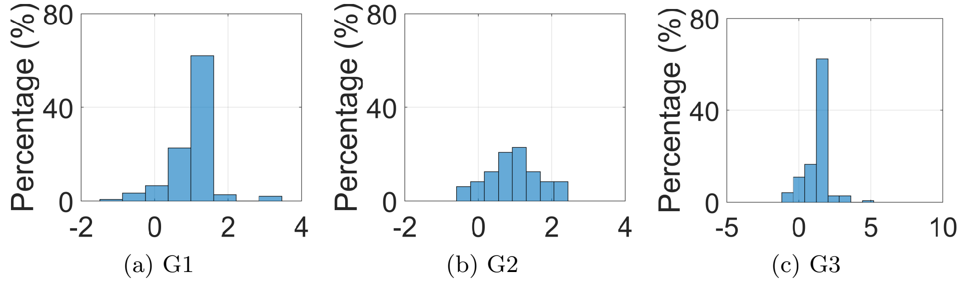


Figure 4.14: Normalized Scanpath Saliency

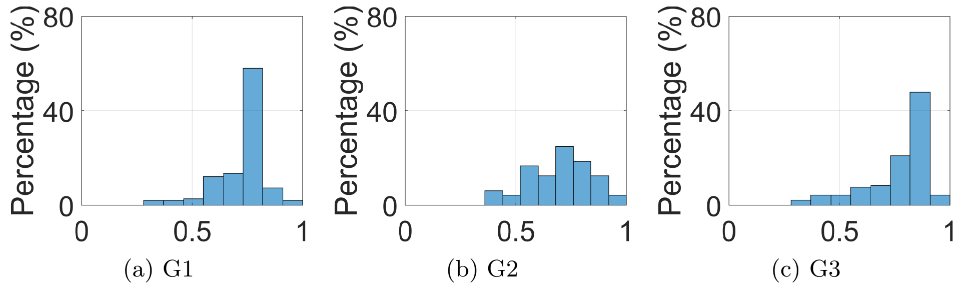


Figure 4.15: Area Under the Curve (Borji)

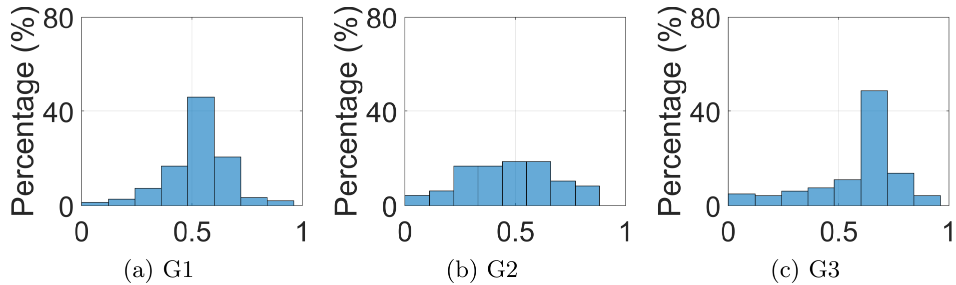


Figure 4.16: Area Under the Curve (Judd)

prediction performance in G2 is worse than others, because the time intervals between two touch interactions are longer than those in G1 and G3. That is, the time intervals between consecutive touch interactions can affect the similarities of gazing patterns among different viewers.

4.8.2 Trace-driven Simulation

We evaluate our interaction-aware optimization framework through trace-driven simulations and a user study under various settings. We first connect a Samsung Galaxy S5 with Android 4.4.2 to a PC (DELL Optiplex 7010) and gain privileged control using rooting tools, CF-Auto-Root. Then, we collect the communication data using *Android tcpdump* and retrieve the tile files using *wget* on this S5 in a campus network. To measure the energy consumption, we supply the power of this S5 using a Monsoon power monitor, which connects to a PC

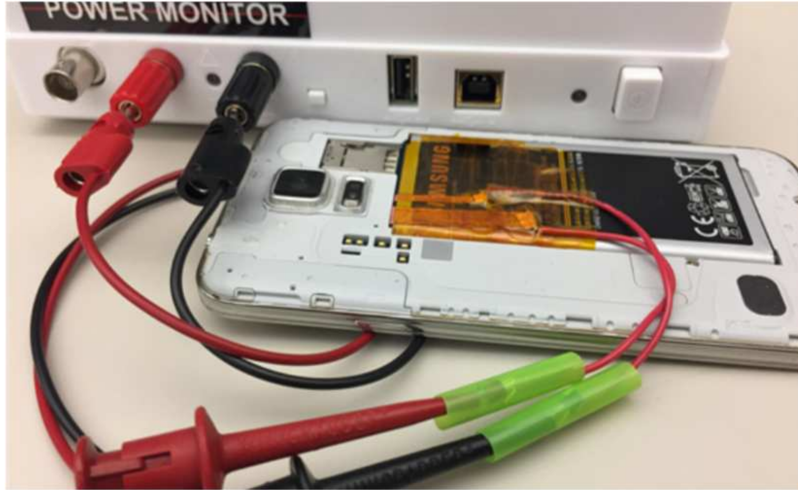
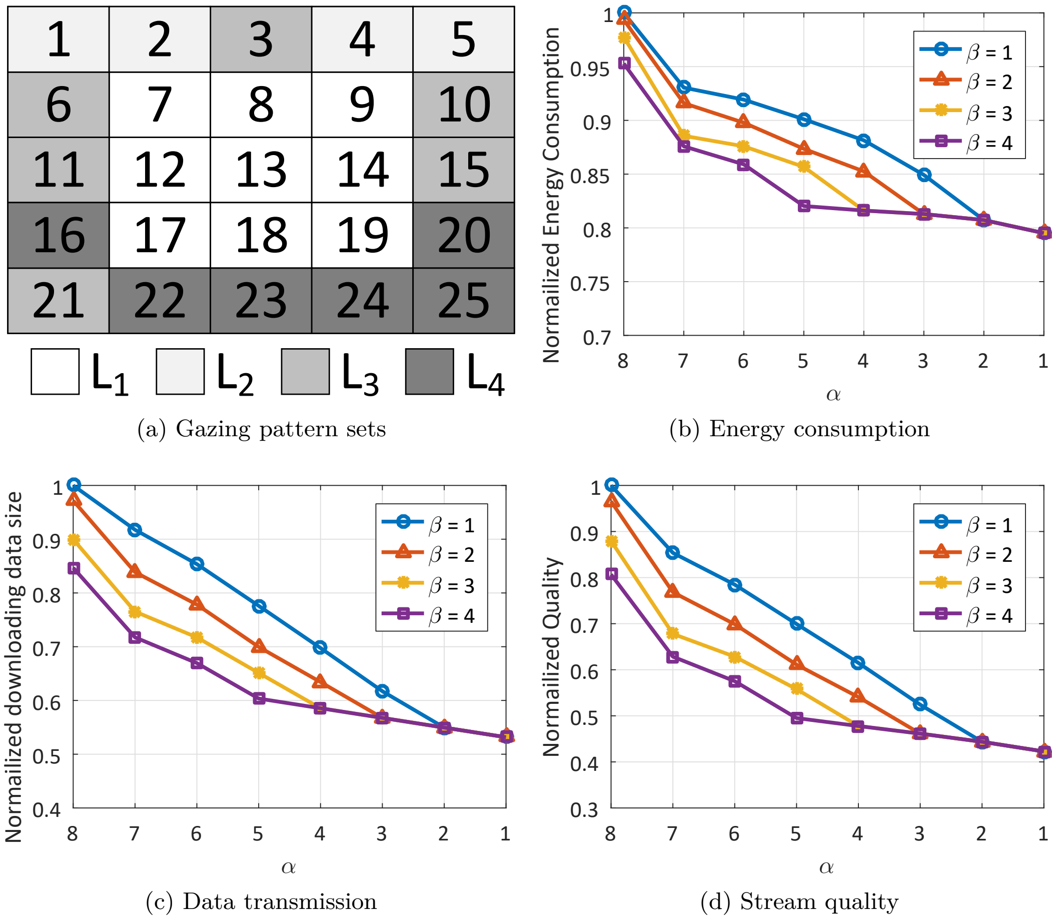


Figure 4.17: Energy measurement platform

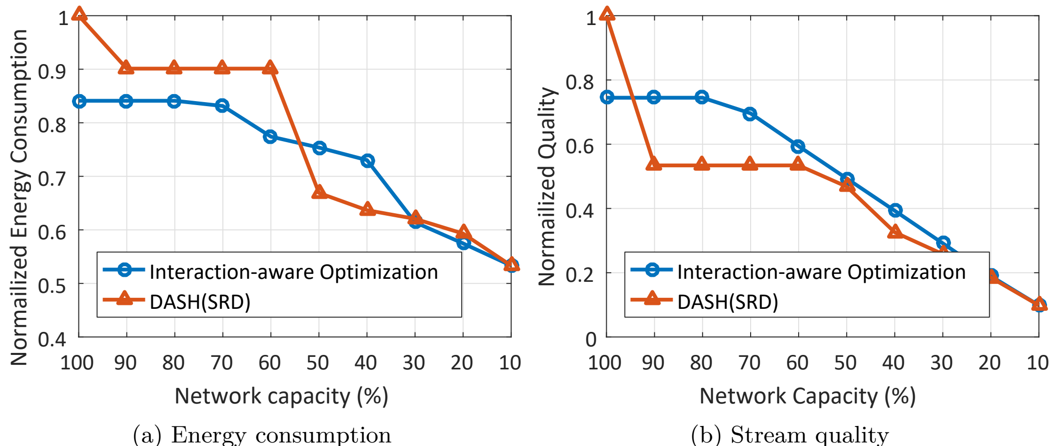


(a) Gazing pattern sets (b) Energy consumption

(c) Data transmission (d) Stream quality

Figure 4.18: Impact of different α and β settings

through USB and feeds back the energy consumption of the viewer's S5 to the PC in real-time, as shown in Figure 4.17.



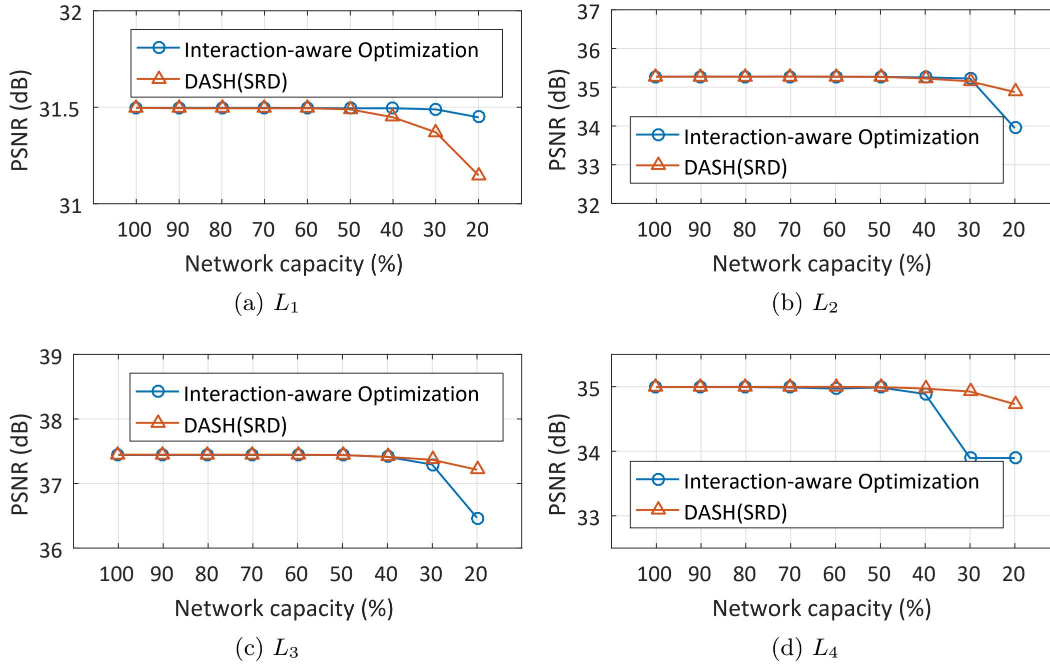
(a) Energy consumption

(b) Stream quality

Figure 4.19: Comparison of our approach and DASH (SRD)

We first investigate the impact of parameters α and β in terms of energy consumption, data transmission, and stream quality. We divide a 2-second video into 25 tiles, which are encoded at eight versions. Based on the predicted gazing pattern sets, as shown in Figure 4.18a, we conduct the tile-based optimization to determine which tile should be obtained and collect the corresponding results, i.e., the energy consumption, data transmission and stream quality, under different parameter settings. For ease of comparison, the results are normalized by the respective maximum values. We plot the normalized results under different α and β settings in Figure 4.18b, 4.18c, and 4.18d. We observe that α has a higher impact than β; therefore, if there is no feasible solution, the optimization algorithm first adjusts β, and then changes α (lines 10 to 16 in Algorithm 4). To avoid the impact of large β, we adjust it only once for different α in Algorithm 4.

To explore the effectiveness of our solution, we also examine the performance of the tile-based optimization under different network capacities through throttling the bandwidth on the mobile device. We compare our method with the original DASH (SRD) selection, in which all tiles have the same version in a short content. In our simulation, we set $\alpha = 8$ and $\beta = 1$ by default. Figure 4.19 plots the results. In Figure 4.19a, we observe that our method has lower energy consumption except for two data points at 50% and 40% of the network capacity. The reason is that the original DASH adaptation may reduce the quality of tiles suddenly to accommodate a decrease of bandwidth, while our method fully utilizes the available bandwidth to optimize the tile quality per energy consumption. Figure 4.19b shows that our solution optimizes the tile quality selection with low energy consumption except for the case of 100% of network capacity, which is determined by our optimization objective, i.e., efficient energy utilization.



(a) L_1 (b) L_2 (c) L_3 (d) L_4

Figure 4.20: PSNR experiments in four gazing pattern sets

4.8.3 User Study

We further conduct a user study to examine the QoE of ten viewers. We first select a 10-second video clip and generate 25 tiles at 8 versions using *ffmpeg*, *x264 encoder*, and *mp4box*[10]. According to the corresponding touch interactions, we predict the gazing pattern sets and output the video clips under different network capacities. We also produce DASH (SRD) video clips for comparisons.

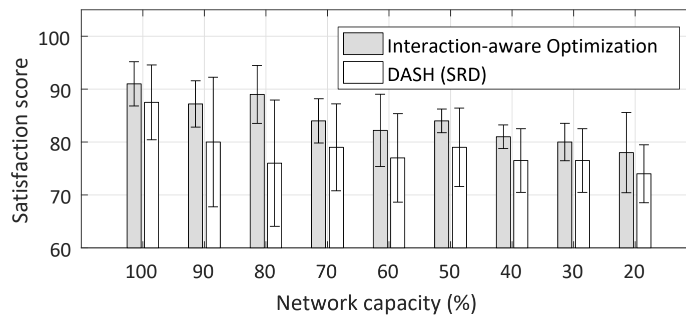


Figure 4.21: Satisfaction Score (error bars are 95% confidence intervals)

We plot the PSNR for each gazing pattern set in Figure 4.20. As can be seen, our approach scores a slightly high PSNR under different network capacities in Figure 4.20a; however, it decreases PSNR in other three sets because our approach optimizes the tiles in the area-

[10]https://gpac.wp.mines-telecom.fr/mp4box

fixation pattern set. To compare the quality differences of the two video clips, we deploy a desktop PC to simultaneously play them under the same network capacity. To guarantee the fairness of comparison, we play the two clips in randomly selected windows every time. The viewers compare their quality differences with the source video clip through grading their satisfaction. Here, we use a satisfaction score to represent the viewers' evaluation about the qualities of their gazing tiles and the whole scene. The satisfaction score is from 1(worst) to 100(best). If they find the quality of one gazing region in video clip A is higher than that in video clip B, they will assign a high score to A. Because the source video clip has the best quality, we assume that its satisfaction score is 100. Figure 4.21 shows the evaluation results under different network capacities. From this figure, we observe that our approach always achieves a higher score (3%-13%) than DASH (SRD).

4.9 Summary

In this chapter, we explored the emerging mobile gamecasting systems in which both stream sources and receivers are mobile devices. Through collecting traces from gamers and viewers in the real world, we identified the relations between the touch interactions of the gamers (i.e., broadcasters) and the gazing patterns of the viewers. Motivated by this, we proposed an interaction-aware optimization framework that includes two novel designs: (1) a touch-assisted prediction module to build association rules offline and performs viewers' gazing pattern prediction online; and (2) a tile-based optimization module for energy consumption and quality selection under limited network capacity. The experimental results showed that our solution effectively utilizes the available bandwidth with better tile quality and less energy consumption. The user study also proved that the approach improves the viewers' satisfaction (from 3% to 13%).

Chapter 5

Highlight-Aware eSports Gamecasting with Strategy-Based Prediction

Fueled by today's high-speed networks and blossom games market, crowdsourced gamecasting has emerged as a popular live streaming service, in which gamers (i.e., streamers) broadcast their game playthroughs[1] in real-time to a large number of viewers who discuss game-related topics at the same time. Such services as Twitch, YouTube Gaming, and DouyuTV[2], have contributed to a significant amount of today's Internet traffic and got into fellow viewers' daily life. As the mainstay of these services, eSports gamecasting channels continuously attract much attention from both gamers and viewers. According to the statistics from Twitch, more than 60% of concurrent viewers are contributed by eSports gamecasting channels every day, such as Dota2, League of Legends (LoL), and CS:GO[3]. Similarly, DouyuTV, the twenty-eighth of top sites in China, also reported that more than 70% of the most attractive streamers are interested in LoL game matches[4].

Compared with professional broadcasting services, e.g., HBO[5] and Apple TV[6], eSports gamecasting services do not host content sources. The encoded game scenes are continuously streamed from content sources managed by gamers to ingesting servers through proprietary

[1]Game playthrough is the act of playing a game from start to finish.

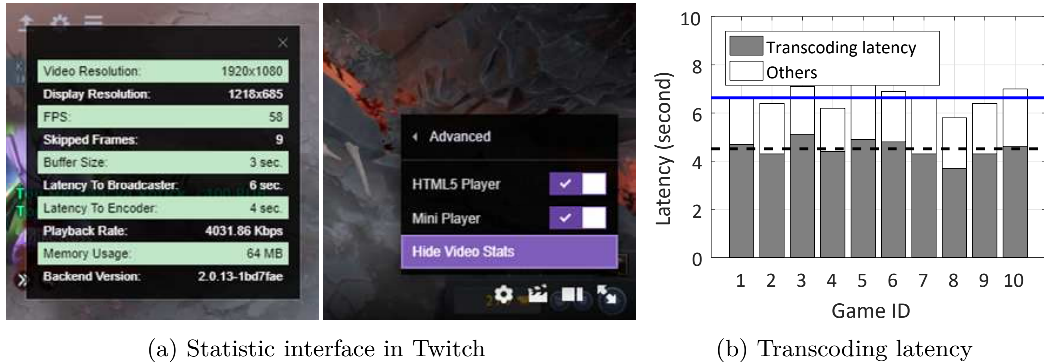
[2]`https://www.douyu.com`

[3]Dota2 and League of Legends are two multiplayer online battle arena (MOBA) video games. Counter-Strike: Global Offensive (CS:GO) is a multiplayer first-person shooter video game.

[4]`https://www.douyu.com/directory/rank_list/game`

[5]Home Box Office (HBO) is an American premium cable and satellite television network. `http://www.hbo.com`

[6]Apple TV is a digital media player and microconsole. `https://www.apple.com/tv`



(a) Statistic interface in Twitch (b) Transcoding latency

Figure 5.1: Impact of transcoding stages

protocols, e.g., RTMP (Real Time Message Protocol[7]). Considering the overhead on the server side, such proprietary protocols cannot be used to deliver gamecasting contents to a large number of viewers [69]. The general way is to transcode original RTMP streaming to HTTP-based streaming, e.g., HTTP Live Streaming (HLS[8]), but consume a massive amount of computational resources and costs [58]. Besides, eSports gamecasting services encourage streamers and fellow viewers to participate real-time discussions. If there exist the huge difference of latencies among fellow viewers, viewers' Quality-of-Experience (QoE) will be largely impaired. For example, a high-latency viewer may know the final result of an eSports match from the low-latency viewers' online discussion before watching the corresponding gamecasting content.

There have been some efforts addressing the aforementioned issues from both industry and academia. Yet the existing solutions either change transcoding settings on the streamer side or optimize the streaming delivery using the information on the viewer side. Such strategies, when used in eSports gamecasting services, lack full knowledge of game events, which lead to the lag and inaccuracy of optimizations. For instance, a strategy first predicts that the highest concurrent viewing number will occur after two minutes in an eSports gamecasting. Then, the corresponding optimization strategy reserves extra computational and bandwidth resources to serve this peak workload in the future. Yet this match and gamecasting are terminated after one minute, the optimization fails and the reserved resources also become wasteful.

To explore the challenges and opportunities for addressing these problems, we first collect the official latency statistics from top-10 streamers of top-10 popular gamecastings in Twitch, as shown in Figure 5.1a, to investigate the characteristics of broadcasting latency, which is defined as the time lag between a game scene on the gamer side and the correspon-

[7]https://www.adobe.com/devnet/rtmp.html

[8]https://developer.apple.com/streaming

ding gamecasting content received on the viewer side. We observed that more than 68% of the broadcasting latency is derived from the transcoding period, as shown in Figure 5.1b. We further analyze transcoding latency by deploying a crowdsourced gamecasting service on various Amazon EC2 instances. Our measurement results reveal that game highlights, i.e., key events in a game, largely impact the complexities of game scenes, which in turn determine the transcoding latencies of corresponding game streams. Motivated by these observations, we propose a novel gamecasting framework *StreamingCursor* to capture the unique features in eSports games, i.e., the multiplayer's real-time operations and strategies, to predict the game highlights. The key challenges towards designing the framework lie in: (1) how to reduce the prediction lag and improve the accuracy; (2) how to adopt the prediction results to optimize the transcoding tasks cost-effectively.

To this end, StreamingCursor incorporates a strategy-based prediction (SBP) module and a highlight-aware optimization (HAO) module. The former predicts the highlights in a game using gamers' interaction data based on a proposed deep learning network, and the latter optimizes the transcoding tasks to reduce the broadcasting latency according to the predictions from the SBP module. To the best of our knowledge, our study is the first to explore the potentials from the multiplayer's operations in the eSports gamecasting services. Our contributions can be summarized as follows: (1) We conduct the cloud-based measurements to reveal the challenges and motivate our design; (2) To usher the optimizations in transcoding and delivery, the framework is designed with full knowledge of multiplayer's game strategies; (3) The performance evaluation demonstrates that the SBP module achieves more than 90% accuracy in the highlight prediction based on the collected replays in the popular eSports game Dota2. In addition, the trace-based simulations show that the proposed optimization approach reduce the broadcasting latency cost-effectively.

5.1 Background

Crowdsourced gamecasting has emerged as one of the most popular live streaming applications in recent years. It offers two parallel services: *Streaming Service* and *Interactive Service*, as shown in Figure 5.2. Game scenes are first captured and encoded by a streaming software (e.g., OBS) deployed on the streamer's device. Then, the streaming service ingests game sessions, assigns transcoding tasks, and distributes contents to viewers. In the meantime, the interactive service provides an embedded chatting platform for the streamers and viewers, creating lots of novel streaming scenarios, e.g., crowdsourced gaming *TwitchPlaysPokemon*. According to the statistics from Alexa[9], Twitch's global traffic ranking just experienced a huge boost from the first hundred to forty-fifth in the last year. Moreover, crowdsourced gamecasting applications also occupy more and more viewers' daily life. For example, "average

[9]`https://www.alexa.com`

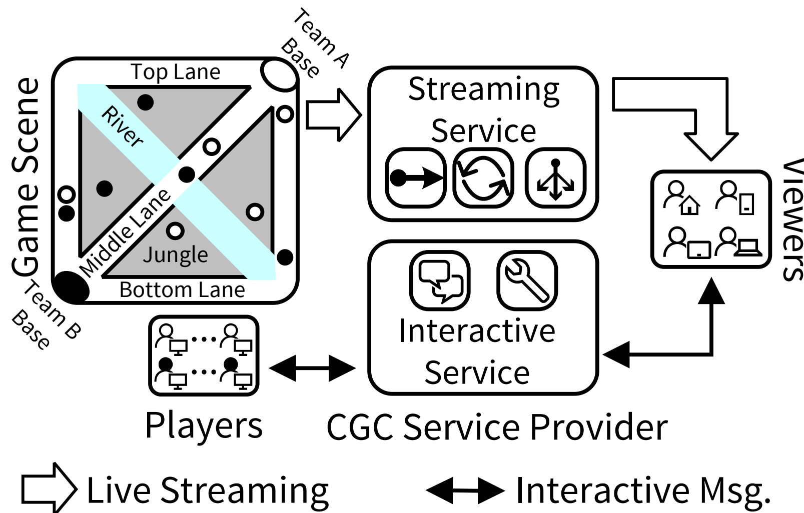


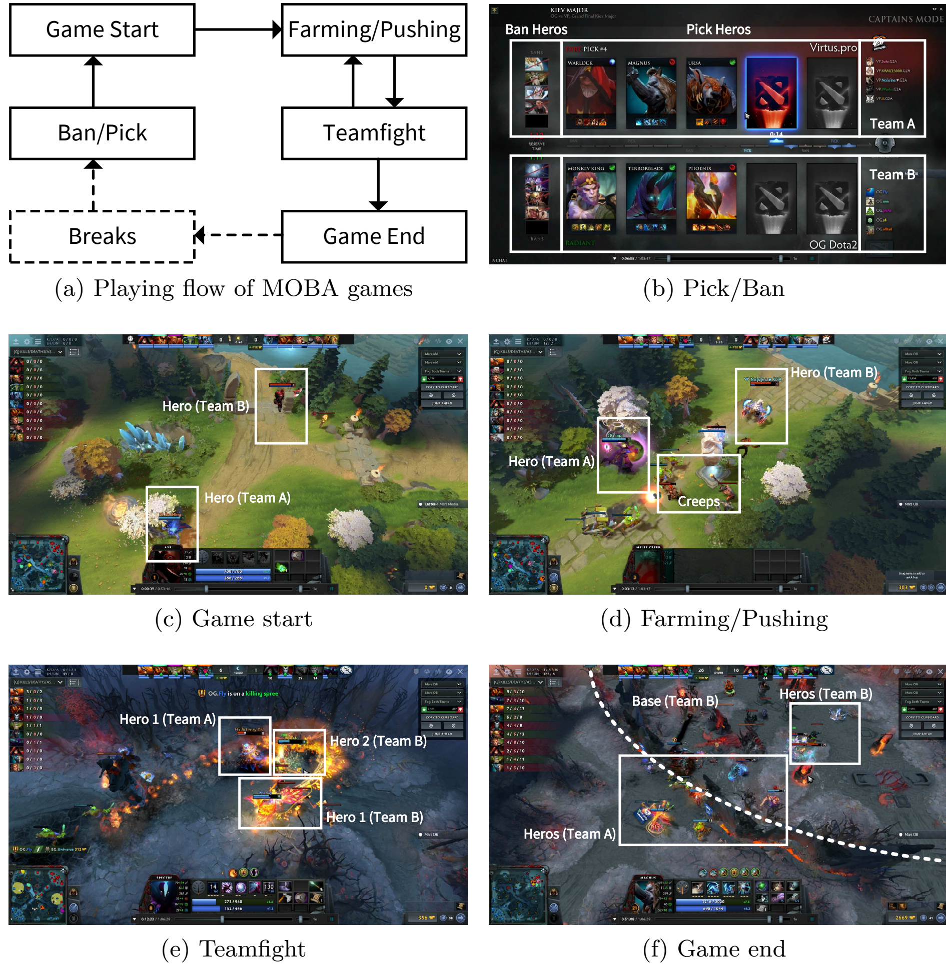
Figure 5.2: An illustration of CILS paradigm

time on site" of Twitch's viewers is about 6 minutes per day, and this index on DouyuTV is about 9 minutes.

As one of the most important branches in such crowdsourced gamecasting applications, eSports gamecasting has attracted millions of viewers from all over the world. It has been reported that global games market generated $91 billion in revenues with a nearly 23% growth in 2016[10], which further elevates the growth of eSports gamecasting. Taking a Twitch's streamer "eleaguetv" as one example, its eSports gamecasting on last January attracted more than 1 million concurrent viewers[11]. As the source of eSports gamecasting, a typical eSports match contains several gamers who are divided into different teams to compete various resources, e.g., golds, weapons, and controlled areas. The left part in Figure 5.2 briefly demonstrates the design of an eSports game "Dota2", in which ten gamers are grouped into two teams. The battle area consists of several types of regions: three lanes (top, middle, and bottom), a river, four jungle areas, etc. When a match starts, every gamer first chooses one hero, then upgrades the hero's level and equipment by fighting with the enemies and the other gamers in another team. Like most of the eSports games, Dota2 also highlights the gamers' cooperation in one team, therefore, how to fight enemies together, called "teamfight", becomes the most important event, which may determine the final result of a match. In a Dota2's teamfight, most of the gamers participate in casting their heroes' skills and attacking others, which also attracts the attentions and discussions from fellow viewers during the gamecasting. Several initial works to detect the highlights and winning team in an eSports match have been studied in recent years. After parsing the replay or

[10]https://www.superdataresearch.com/insights

[11]https://goo.gl/63iNrK



(a) Playing flow of MOBA games (b) Pick/Ban

(c) Game start (d) Farming/Pushing

(e) Teamfight (f) Game end

Figure 5.3: Illustrations of the playing flow in Dota2.

video of a match, these studies focus on the changes of the gamers' information, e.g., location, gold, and experience, and the dynamics of visual effects in the videos. Yang *et al.* [76] analyzed the replays using a sequence of event-related graphs and extracted key patterns to predict the successful team of the entire game. They achieved an 80% prediction accuracy when testing on new game replays. Drachen *et al.* [18] examined the spatiotemporal patterns of the gamers with four skill tiers, i.e., normal, high, very high, and professional. They further analyzed the relationship between game skills and match results. The work in [65] detected the highlights from the videos of eSports game matches. By using CNNs to learn the features of visual effects in the videos, the proposed approaches achieves more than 80% accuracy in the highlights detections. Similarly, Chu *et al.* [14] recognized the designated text displayed on a game screen when key events occur as visual features. The proposed highlight prediction models are based on the visual features, event features, and viewer's behavior. In addition, their approach also can predict the emerge of a highlight in the next few seconds.

These previous efforts motivate our work, but still face two limitations: (1) because the highlights only can be detected after they appear, existing approaches cannot meet the timeliness requirement of the optimization in eSports gamecasting applications; (2) based on pre-recorded eSports videos, current prediction strategies cannot better guide the optimization due to the low accuracy and short forecasting gap. To overcome these limitations, we directly investigate the interactions among the gamers in an eSports game, closely explore the correlations between these interactions and highlights, and timely predict the highlights.

5.1.1 Optimization in Crowdsourced Gamecasting Services

Some recent studies already focused on the optimization in crowdsourced gamecasting services. Fan-Chiang *et al.* [76] investigated the importance of segment-of-interest in such services and optimized the resource allocation during the gamecasting. The proposed framework reduces the consumption of bandwidth resource and improves the viewing quality. Wu *et al.* [86] explored opportunities to combine the edge-based and cloud-based network resources. A novel framework to allocate the resources was designed to improve the ingesting performance in crowdsourced gamecasting services. He *et al.* [27] proposed a framework "CrowdTranscoding" for crowdsourced gamecasting services through allocating the transcoding assignment to the massive viewers. Both the trace-driven simulation and experiments show the superiority of this novel framework. Our work complements them by exploiting the hints that guide the optimization in crowdsourced gamecasting services in advance, even before the encoding stage of a gamecasting. Our work proposes a new design in the realm of utilizing the gamers' interactions in eSports games, where the cooperation and competition data are extracted into gamer-related, team-related and item-related features. Integrating these features into deep learning models pose a new opportunity to optimize the eSports gamecasting services and improve the viewers' QoE.

5.1.2 Learning-based QoE Improvement in Multimedia Systems

Learning-based approaches have recently become popular solutions to improve viewers' QoEs in multimedia systems. In a typical learning-based approach, a prediction module collects the feedbacks from streaming servers and viewers and forecast the viewers' QoE in various locations and network connections. As shown in [36], the proposed data-driven prediction model coordinates the relationships between video quality and observed session features, capturing the key features and updating quality predictions in near real-time. Vega *et al.* [68] used unsupervised deep learning techniques and measurements on the viewer side to accurately assess the video quality in real-time. Jiang *et al.* [37] introduced a prediction-based mechanism, considering the video sessions that share the same features into a group. The proposed framework well responds the quick changes and biases in the real-time QoE optimization. Our work differs from these studies in two aspects: First, we

Table 5.1: Configuration of Amazon EC2 instances

ID	Type	vCPU	Memory (GB)	Price
A1	m4.xlarge	4	16	$0.2/h
A2	m4.2xlarge	8	32	$0.4/h
A3	m4.4xlarge	16	64	$0.8/h
A4	c4.xlarge	4	7.5	$0.199/h
A5	c4.2xlarge	8	15	$0.398/h
A6	c4.4xlarge	16	30	$0.796/h

employ the feedbacks from gamers and broadcasters in crowdsourced eSports gamecasting services, rather than the sessions features from various viewers, which is a significant difference from a system point of view. Second, the interactions of gamers are ahead of the viewing behaviors and sessions on the viewer side, which guarantees the timeliness of the prediction results in our work.

5.2 Measurement and Observation

In this section, we investigate the characteristics of transcoding tasks on various Amazon EC2 instances. The measurements are mainly based on the game Dota2, which is a typical multiplayer online battle arena (MOBA) game and one of the most popular eSports games. The measurement results also motivate our work in this section.

5.2.1 Measurement Settings

In a Dota2 match, the game flow mainly includes the following five parts, the corresponding snapshots are shown in Figure 5.3:

- Pick/Ban (P/B): Pick/Ban stage is the first phase of a MOBA tournament match. The captain in each team bans certain heroes, preventing either team from picking the hero; every captain also chooses five heroes for the whole team alternately;

- Game Start: After choosing preferred heroes, the gamers in a team start the game from their base and choose the combat lanes according to the heroes' abilities;

- Farming/Pushing (F/P): In this stage, all gamers upgrade their equipment and levels through earning golds and experiences from clearing enemy creeps on the lanes or combating the neutral creeps at the jungle areas;

- Teamfight (TF): During an eSports gamecasting, teamfights are the most attractive events, in which the gamers attack the opponent heroes using their skills. It occurs in several farming/pushing stages. Because the goal of a teamfight is to eliminate as many of the opponent heroes as possible, a good teamfight can determine the final result of a match.

- Game End: After several rounds of farming, pushing and teamfight, the gamers in a team will try to attack the opponents' base, if success, they will win the match;

Our measurement for each stage relies on ten videos that are recorded from five randomly selected Dota2 replays. we conduct the measurements on Amazon EC2 instances. As shown in Table 5.1, we consider six types of instances, which have various settings and prices. A1, A2, and A3 are m4 instances, which can provide a balance of compute, memory, and network resources. A4, A5, and A6 are c4 instances, which are optimized for compute-intensive workloads and delivers very cost-effective high performance at a low price/compute performance in EC2. We measure the transcoding performance using FFmpeg to convert the original videos to different resolutions and bitrates. The broadcasting latency is measured by deploying Nginx-RTMP module[12] on each platform. The instance with this module can be considered as a crowdsourced livecast server that ingests video contents from a source and transcodes it to various quality versions.

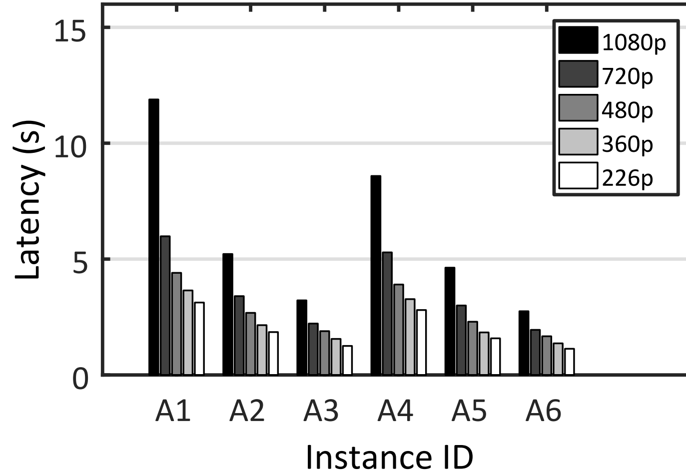
5.2.2 Measurement Results

Figure 5.4 shows the measurement results for the transcoding tasks on different instances. As shown in Figure 5.4a, we observe that the computation-optimized instances A4, A5, and A6 achieve lower latencies than the general instances A1, A2, and A3, even they have the similar cost. But the superiority of computation-optimized instances is decreased in the transcoding tasks with the low-resolution quality versions. Figures 5.4b indicates the performance of transcoding the game scenes in different stages on A4. We find that the latency of P/B stage is clearly lower than the other stages. Besides, the transcoding tasks in TF and End stages need more time to process complex game scenes, which introduce extra latencies. We also observe the huge disparity among different streaming qualities. For example, in TF stage, the average transcoding latency of the 1080p videos is 3 seconds higher than that of the 720p videos in Figure 5.4b. Figure 5.4c shows the impact of the number of the threads on A6 instance. Through changing the number of the used CPU threads to a reasonable setting, we can dynamically adjust the transcoding latencies for the different quality versions in a gamecasting. These measurement results motivate us to design a framework that not only predicts the highlights in a gamecasting but also balances the transcoding latency and cost in different settings according to the prediction results.

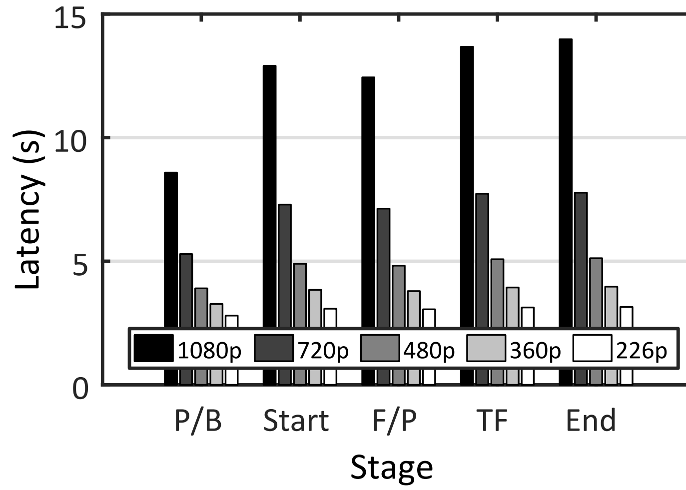
5.3 StreamingCursor: An Overview

In this section, we first outline the StreamingCursor framework and proposed the detailed design in the following two sections. As shown in Figure 5.5, StreamingCursor is a generic

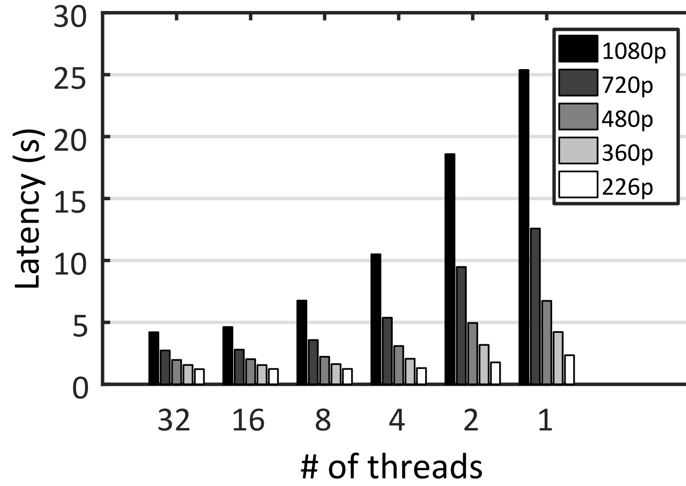
[12]`https://github.com/arut/nginx-rtmp-module`



(a) Impact of different instances (P/B stage)



(b) Impact of different stages(A4)



(c) Impact of different thread settings (TF stage on A6)

Figure 5.4: Measurement results for the transcoding tasks on different instances

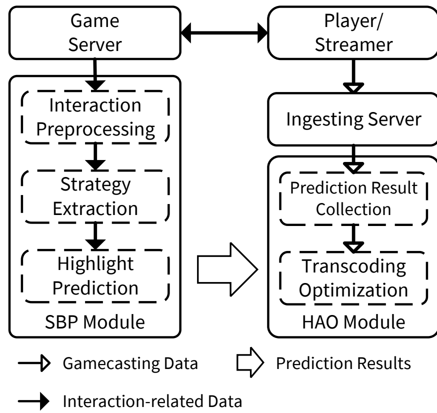


Figure 5.5: Workflow of the framework StreamingCursor

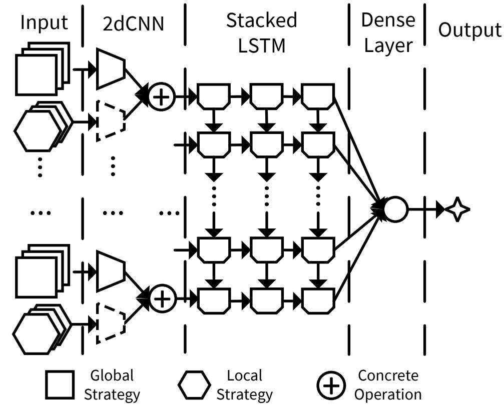


Figure 5.6: Design of highlight prediction

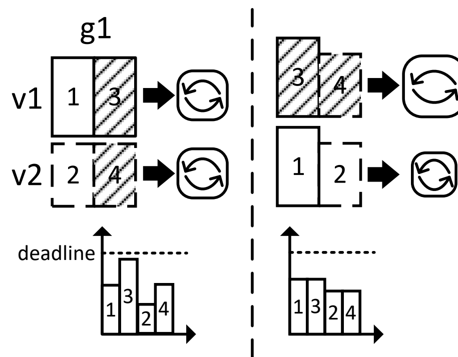


Figure 5.7: Example of task assignment

framework that facilitates the optimization of existing eSports gamecasting services. It consists of two modules, namely, Strategy-based Prediction (SBP) module and Highlight-aware Optimization (HAO) module. The SBP module is a new part of this framework compared with that of existing eSports gamecasting services. Upon receiving the gamers' interactions in a match from the game server in real-time, the SBP module will be aware of the team strategies, predicting the highlights, e.g., teamfights, in this match. The HAO module receives the predictions from the SBP module as the hints to guide the transcoding and distribution of gamecasting. Combining these hints and the viewing demands, the HAO module reserves and allocates resources ahead of the emerges of the predicted highlights cost-effectively.

There are however a number of critical practical and theoretical issues to be addressed in this generic framework. First, the gamers' interactions change over time, which generates the varying strategies of cooperations in a team and competitions between two teams. Although the existing study in [14] have predicted the highlights in the next few seconds, the accuracy of its prediction is not very high. As such, StreamingCursor must well forecast the highlights with a higher accuracy and a larger prediction gap after receiving the

Table 5.2: Statistics of a replay file

Type	Description	# of logs	Percentage
4	Hero's movements	392,268	87.9%
24	Damage in combats	29,544	6.6%
3	Hero's local statistic, e.g., gold and experience	5,810	1.3%
27	Ability used in combats	1,801	0.4%
26	Death in combats	3,903	0.9%
29	Item buy/sell	600	0.1%
others	Other info. in combats, gamers' statistics, etc.	12,434	2.8%

interactions. Longer predicted gap, earlier to prepare the corresponding optimization. In addition, after predicting the highlights, how to cost-effectively allocate the transcoding tasks and strategically design the content distribution will significantly impact the viewers' QoE. These problems are further complicated given the heterogeneous devices and network connections of viewers and that of streamers (i.e., gamers). Besides, such a framework should be transparent to the streamers and viewers, i.e., the whole design does not suspend the match, impact the gamecasting, or reduce the viewers' QoE. Therefore, it can complement with existing eSports gamecasting services.

5.4 Framework Design and Solution

In this section, we propose the design and solution for the highlight prediction in eSports matches and the gamecasting optimization. We first illustrate the strategy-based prediction (SBP) module based on gamers' interactions.

5.4.1 Strategy-based Prediction

MOBA games usually save game matches by recording the gamers' interactions into the proprietary replay files. We, therefore, are able to retrieve every gamer's interaction data through parsing these game replays. To better understand these interactions and preprocess them, we first classify them into several groups according to their interaction types that are pre-defined by the game developers. As shown in Table 5.2, we show the details of different types of data in a replay. We observe that about 88% of data show the changes of gamers' locations, and 6.5% are combat logs, which illustrate how gamers attack their enemies, e.g., damage values during a combat. Besides, other data consist of the changes of gamers' golds, experiences, skill levels, equipment, etc. These interactions naturally can be further classified into spatial and temporal interactions: (1) spatial interactions, e.g., every gamer has a fixed location at a certain time; (2) temporal interactions, e.g., the golds owned by every gamer will be changed along with the time of a match. The StreamingCursor

framework, therefore, must capture the characteristics of these two types of interactions. To fulfill this target, we extract global data and local data. The global data contain the location changes of all gamers, the unit changes of the whole battle arena, and the competition data of two teams. The local data includes the changes of the gold, experience, buy/sell items of every gamer along with match time.

Finally, the SBP module is mainly constructed by a deep learning network, as shown in Figure 5.6. It consists of several 2dCNN (Convolutional Neural Network) layers [41, 17] and stack LSTM (Long Short-Term Memory) layers [29], capturing the correlations between the spatiotemporal features in an eSports game and the highlights in the corresponding gamecasting. In deep learning research, the CNN is a class of deep, feed-forward neural network that has successfully been applied to processing videos/images and decreasing the dimensions of input data. LSTM is a type of RNN (Recurrent Neural Network) [52], well classifying, processing and predicting time series. Through unique gating units, LSTM avoids the long-term dependency problem, having the ability to remove or add information when learning time series.

In the SBP module, the first part is two 2dCNN layers for global strategy and local strategy, respectively. For each time slot, the 2dCNNs can better reduce the dimensions of two types of input data. Then, we concrete the outputs of two 2dCNNs as the input of the stacked LSTM Networks. The stacked LSTM layers output the prediction results, indicating that the highlights will occur. To reflect the prediction lag, we train the model using different input data labeled by various time gaps.

5.4.2 Transcoding Task Assignment Optimization

In this subsection, we design the transcoding optimization. We assume a general transcoding framework, where a large number of transcoding tasks need to be assigned to different servers with various computational configurations. Because highlight predictions give us opportunities to reserve and allocate resource earlier, we perform the transcoding task assignment according to the following two rules: (1) the workloads that belong to a predicted highlight should be transcoded priority-wise, such that the transcoding time for these workloads can be reduced using high-performance servers; (2) other workloads should be finished before its deadline when viewers request them, such that our optimization does not impair viewers' QoEs. Figure 5.7 illustrates the basic concept of transcoding task assignment optimization. Gamecasting $g1$ needs to be transcoded into two versions: $v1$ and $v2$. Based on the predictions, we already know the time when a highlight occurs (shadow areas); therefore, we first assign the workloads 3 and 4 on the high-performance server s1, and then assign the workloads 1 and 2 on the low-performance server $s2$. Finally, all transcoding tasks will be finished before their deadlines and the total transcoding latency is lower than the previous assignment. Yet how to allocate these tasks cost-effectively still need to be addressed. Next, we formulate this problem and propose the solution.

We denote (g, v) as a task that transcodes a gamecasting g from the original quality version to quality version v. Without loss of generality, we use $S^{(T)}_{(g,v)}$ to denote the transcoding server that is assigned to task (g, v). Let $E^{(T)}[(g, v), s]$ denote the transcoding cost if task (g, v) is allocated to server s in time slot T. It can be calculated as follows:

$$E^{(T)}[(g, v), s] = \frac{c^{(T)}(g, v)}{\mathbb{C}(s)}\mathbb{P}(s)$$

where $c^{(T)}(g, v)$ is the amount of computation resource that is required by task (g, v) in time slot T. $\mathbb{C}(s)$ and $\mathbb{P}(s)$ are the computation capacity and unit price of a server s, respectively. We also denote the profit of transcoding task (g, v) on server s in time slot T as $F^{(T)}[(g, v), s]$, which can be calculated as follows:

$$F^{(T)}[(g, v), s] = W^{(T)}(g, v) log(\alpha - \beta l^{(T)}[(g, v), s])$$

where $W^{(T)}(g, v)$ is the predicted viewing number of the version v of gamecasting g; $l^{(T)}[(g, v), s])$ is the transcoding latency of task (g, v) on server s. Base on the definitions of the transcoding cost and profit of tasks, we define the transcoding gain $TG^{(T)}[(g, v), s]$ when assigning task (g, v) on server s as follows:

$$TG^{(T)}[(g, v), s] = \frac{F^{(T)}[(g, v), s]}{E^{(T)}[(g, v), s]} \tag{5.1}$$

As such the optimization of transcoding tasks in time slot T is then formulated as follows:

$$\max \sum_g \sum_v TG^{(T)}[(g, v), S^{(T)}_{(g,v)}] \tag{5.2}$$

subject to:
Transcoding Latency Constraint:

$$\begin{aligned} l^{(T)}(g, v)H^{(T)}_{(g,v)} &\leq \frac{\sum_{t\in[0,T-1]} l^{(t)}(g, v)(1 - H^{(t)}_{(g,v)})}{\sum_{t\in[0,T-1]}(1 - H^{(t)}_{(g,v)})} \\ &\leq \gamma, \forall (g, v) \in G^{(T)} \end{aligned} \tag{5.3}$$

$$H^{(T)}_{(g,v)} = \begin{cases} 1, & \text{if task } (g, v) \text{ belongs to a highlight} \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

Resource Availability Constraint:

$$\sum_g \sum_v c^{(T)}(g, v) \leq \sum_s C^{(T)}(s) \tag{5.5}$$

where $H_{(g,v)}^{(T)}$ shows whether task (g, v) belongs to a highlight, as shown in equation (5.4). The optimization is to assign the transcoding tasks to different servers, so that the overall transcoding gain can be maximized. The rationale of transcoding constraint is as follows: (1) the tasks that belong to highlights should enjoy a similar transcoding time compared with other tasks; (2) all tasks should be completed in a period γ, which proposes a basic QoE requirement for the transcoding tasks in eSports gamecasting services. The resource availability constraint guarantees that there exist enough computational resources for all transcoding tasks.

Because the Transcoding Latency Constraint is independent of the Resource Availability Constraint, we can first remove the transcoding servers that do not meet the former, and then this problem can be transformed into a minimum cost network flow problem. The minimum cost network flow problem [3] is briefly introduced as follows. Let $G = (N, E)$ be a directed network with a set N of n nodes and a set E of m edges. Every node $i \in N$ has an associated number $b(i)$ to denote its supply or demand depending on whether $b(i) > 0$ or $b(i) < 0$. Every edge $(i, j) \in E$ has an associated per unit flow cost c_{ij} and a flow capacity u_{ij}. The minimum cost flow problem can be formulated as follows. Such problem can be solved by double scaling algorithm [3] in polynomial time. Let C denote the largest magnitude of any edge cost and U denote the largest magnitude of any edge capacity. The time complexity is $O(nmlogUlog(nC))$.

$$Minmize \sum_{(i,j)\in E} c_{ij}x_{ij}$$

subject to:

$$\sum_{\{j:(i,j)\in E\}} x_{ij} - \sum_{\{j:(j,i)\in E\}} x_{ji} = b(i), \forall i \in N$$

$$0 \leq x_{ij} \leq u_{ij}, \forall (i,j) \in E$$

To transform our optimization to a minimum cost flow problem, we next introduce several transformation rules:

1. We introduce supply node v_{start} and demand node v_{end}.
2. For each transcoding task (g, v), we add a node $v_{(g,v)}$ and an edge $(v_{start}, v_{(g,v)})$, its capacity $u_{v_{start},v_{(g,v)}}$ is equal to the computational requirement of transcoding task (g, v);
3. For each transcoding server s, we add a node v_s and an edge (v_s, v_{end}), and its capacity $u_{v_s,v_{end}}$ is equal to the amount of available computational resource on server s;
4. If task (g, v) can be assigned to server s, we add an edge $(v_{(g,v)}), v_s)$ and its capacity $u_{v_{(g,v)}),v_s}$ is equal to the computational requirement of transcoding task (g, v).

Table 5.3: Performance of the strategy-based highlight prediction on different networks

Networks	CNN	LSTM	Input Strategy		Game Event	
ID	# of Layers	# of Layers	Global	Local	Highlight	Non-highlight
1	1	1 (Single)	o	o	0.92	0.92
2	1	2	o	o	0.95	0.94
3	1	**3 (our design)**	**o**	**o**	**0.95**	**0.95**
4	1	4	o	o	0.93	0.92
5	2	3	o	o	0.92	0.92
6	1	3	o	x	0.79	0.85
7	1	3	x	o	0.67	0.67

5. We set the cost from v_{start} to any task node $v_{(g,v)}$ and the cost from any server node v_s to v_{end} to 0, and set the cost of existing edge $(v_{(g,v)}, v_s)$ to $-[TG[(g,v),s]/u_{v_{(g,v)},v_s}]$[13].

6. To meet our optimization problem, we use the following constraints in the transformed minimum cost flow problem:

$$x_{v_{start},v_{(g,v)}} = u_{v_{start},v_{(g,v)}}, \forall v_{(g,v)} \in N$$

$$x_{v_{(g,v)}),v_s} = u_{v_{(g,v)}),v_s}, \forall v_{(g,v)}, \forall v_s \in N$$

$$x_{v_s,v_{end}} \leq u_{v_s,v_{end}}, \forall v_s \in N$$

After applying the above rules, we can transform our optimization problem (5.1) into a minimum cost flow problem.

5.5 Performance Evaluation

We have conducted trace-driven experiments to evaluate the performances of the prediction module SBP and the optimization module HAO in the framework StreamingCursor.

5.5.1 Strategy-based Prediction Results

We collect 80 game replays from Dota2. Based on an open source package[14], we write an offline parser to extract interaction data of every replay into continuous time slots and recognize teamfights based on the damage and multiple kill data. Because the SBP module is expected to predict highlights before their occurrences, we assume that the interactions between time slot T and $T + t_{gap}$ will trigger a highlight, i.e., teamfight, if they occur t_{gap} seconds ahead of this highlight. Here, we set prediction gap t_{gap} to 10 seconds. Based on

[13] Without loss of generality, we omit the index time slot T.

[14] `https://github.com/dotabuff/manta`

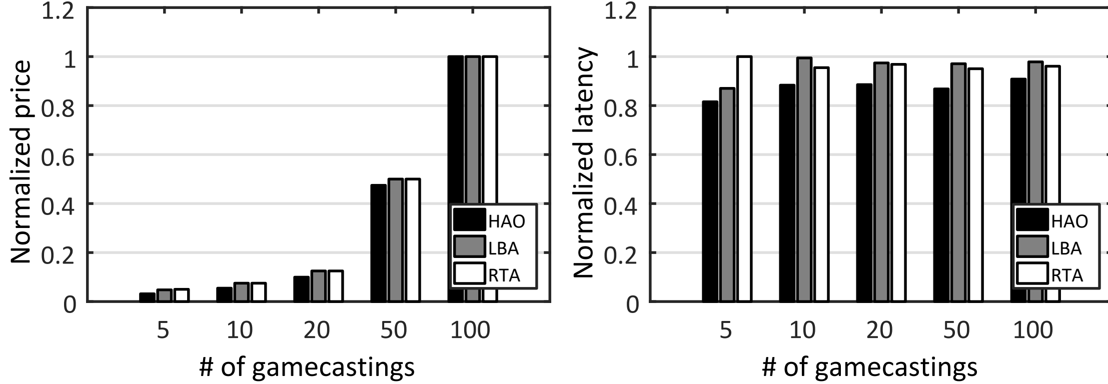


Figure 5.8: Normalized transcoding expense under different number of broadcasters

Figure 5.9: Normalized transcoding latency under different number of broadcasters

this rule, we label the data in different time slots using *highlight* or *non-highlight*. We train the proposed learning network using the half of these data, and then test the prediction performance of the network using the remaining half. The deep learning based highlight prediction in the SBP module is implemented in Keras[15] with cuDNN on an Nvidia GTX 1080 GPU. For the comparison of SBP module, we have also implemented several networks with different settings, as shown in Table 5.3. In our evaluation of this module, we use F-score [25] to reflect the performance of the SBP module.

We first evaluate the impact of the number of the LSTM layers. As shown in the networks 1 to 4 in Table 5.3, we adjust the number of the LSTM layers and fix other settings. It can be seen that network 1 with single LSTM layer has the lowest prediction performance. When we use stacked LSTM and increase the layer number, network 3 acquires better result than networks 1 and 2. We further test a stacked LSTM with four layers and find the prediction performance is lower than networks 2 and 3. We consider that this deeper LSTM design overfits the training data. Based on this observation, we fix the number of stacked LSTM layers to 3 and change other settings to study the impact. In network 5, as shown in Table 5.3, the number of CNN layers is added to 2, but this decreases about 3% of the performance in the highlight prediction. The extra CNN layer, which is not suitable, reduces the dimensions of the training data too much and further impact the effectiveness of LSTM layers. To investigate the impact of global and local strategies in our design, we train networks 6 and 7 using global strategy or local strategy only. As shown in Table 5.3, networks 6 and 7 suffer a significant performance decrease due to the lack of local or global strategies. This illustrates the effectiveness of adopting global and local strategies together.

[15] `https://keras.io`

5.5.2 Highlight-aware Optimization Performance

For the comparison of HAO module, we have implemented two strategies: Randomly Transcoding Allocation (RTA) and Load-Based Allocation (BPA). The former assigns a transcoding task on a randomly selected server from all available ones. The latter always selects the transcoding server with the lowest load for the current task. In the evaluation of this module, we adopt two metrics: transcoding latency and expense, which combined together illustrate the performance when using three strategies. We use the transcoding traces collected in Section 5.2 and the gamecasting traces crawled using Twitch official API in one hour. We write three simulators to conduct the performance evaluation, which can choose transcoding servers from three types of instances, i.e., A4, A5, and A6 and elastically lease/release them. Considering the transcoding settings in practical gamecasting services, we assume that all gamecastings are transcoded from original RTMP streaming to HLS segments with four quality versions, i.e., 1080p, 720p, 480p, and 360p[16], using the instances on Amazon EC2.

Transcoding Cost on the Cloud

In this experiment, because the instances can be leased elastically, they can satisfy all the transcoding requests of gamecastings. We calculate the cost of leasing Amazon EC2 instances as transcoding servers. In Figure 5.8, each bar represents the cost when a particular number of gamecastings are transcoded during one hour. For ease of comparison, the results are normalized by the maximum cost. It is noted that transcoding a large number of gamecastings generally consume larger computation resources. The reason is that all gamecastings have to be processed and transcoded, even if some of them do not have any viewer. For any number of gamecastings in this figure, we observe that the transcoding cost with our HAO module is less than or equal to that with the other two approaches.

Transcoding Latency

In the following experiments, we will evaluate the effectiveness of our HAO module. We focus on the average transcoding latency, which is the average time when transcoding a large number of gamecastings. For ease of comparison, the results are normalized by the maximum latency in Figure 5.9. We evaluate the impact of the different number of gamecastings. As shown in Figure 5.9, each bar represents the average transcoding latency when a particular number of gamecastings are transcoded during one hour. We can observe that our approach can effectively reduce the transcoding latency, with an average 10% decrease. In particular, about 20% of the transcoding latency is reduced when the number of the gamecastings is equal to five.

[16]`https://stream.twitch.tv/encoding`

5.6 Summary

In this paper, we showed the evidence that in practical crowdsourced gamecasting platforms like Twitch, the transcoding complexity and latency of an eSports gamecasting closely depends on the gamers' interactions in the corresponding game match. Motivated by this observation, we explored opportunities and challenges to optimize the transcoding tasks in eSports gamecasting services. We presented *StreamingCursor*, a generic framework that investigates eSports gamers' interactions and predicts game highlights to optimize the corresponding transcoding tasks cost-effectively. Specifically, we first designed a strategy-based prediction (SBP) module, which mainly adopts a deep learning based network to predict highlights in real-time. We further proposed a highlight-aware optimization (HAO) module, which receives the prediction results from the SBP module and select transcoding servers cost-effectively. Extensive simulations driven by traces from Amazon EC2 and Twitch illustrated the high prediction accuracy of the SBP module and the cost-effectiveness and superior transcoding performance of the HAO module.

Chapter 6

Conclusion

Crowdsourced livecast systems have emerged in recent years. Such systems as Twitch, YouTube Gaming, and Periscope have received much attention both from industry and academic and substantially changed the generation and consumption of live streaming content, occupying more and more Internet users' daily life. In this thesis, we have conducted measurement study to investigate the existing crowdsourced livecast systems and proposed enhancements to augment the content ingesting, transcoding, and distribution. In this chapter, we first summarize this thesis, and then discuss the future directions.

6.1 Summary of the Thesis

First, we conducted the measurement study on crowdsourced livecast systems. Taking Twitch as one example, we revealed the views patterns and source/event-driven features. To better understand the challenges and opportunities inside, we further deploy a testbed to examine the streaming uploading and receiving on the broadcaster side and view side, respectively. The results show that (1) the disparity of streaming latencies among different viewers significantly impact their viewing experiences; (2) it is a big challenge for the service providers to host the dynamic broadcasters pose and provide the high-quality ingesting links.

Second, we measured the broadcasters' resource consumption in Twitch, revealing that a large number of unpopular broadcasters occupy the valuable bandwidth and computational resources on dedicated servers. We further proposed a cloud-assisted framework for the initial offloading, as well as dynamic ingesting redirection and transcoding assignment, to migrate crowdsourced live contents between dedicated servers and public clouds. Extensive simulations driven by traces from Twitch and Amazon EC2 demonstrated the superiority of our design under diverse configurations.

Third, we investigated the emerging mobile gamecasting, in which both livecasting sources and receivers are mobile devices. We investigated the associations between the touch interactions of the gamers (i.e., broadcasters) and the gazing patterns of the viewers and

proposed an interaction-aware optimization framework, including two novel modules: (1) a touch-assisted prediction module to build association rules offline and performs viewers' gazing pattern prediction online; and (2) a tile-based optimization module for energy consumption and quality selection under limited network capacity. The experimental results showed that our solution effectively utilizes the available bandwidth with better tile quality and less energy consumption. The user study also proved that the superiority of our approach against the state-of-the-art method.

Fourth, we explored the eSport gamecasting, the most popular branch of crowdsourced livecast systems. Based on the measurement study on Twitch and Amazon EC2, we revealed that the transcoding complexity and latency of an eSports gamecasting closely depends on the gamers' interactions in the corresponding game match. We further presented a generic framework *StreamingCursor* to extract eSports gamers' interactions and predict game highlights to optimize the corresponding transcoding tasks cost-effectively. In this framework, we designed two modules: (1) a strategy-based prediction (SBP) module, which mainly adopts a deep learning based network to predict highlights in real-time; (2) a highlight-aware optimization (HAO) module, which receives the prediction results from the SBP module and select transcoding servers cost-effectively. Extensive simulations driven by traces illustrated the high prediction accuracy of the SBP module and the superior transcoding performance of the HAO module.

6.2 Future Directions

There are still several directions that can be further explored in the future work.

6.2.1 Further Measurements on Crowdsourced Gamecasting

In February 2014, a pilot project "Twitch Plays Pokemon" offered live streaming and game emulator for the game Pokemon Red, in which players (also as the viewers in Twitch) simultaneously send the control message of Pokemon through the IRC protocol and live messages in Twitch. This truly crowdsourced game streaming attracted more than 1.6 million players and 55 million viewers. Similar scales however have yet to appear in other interactive events, though. It is also known that the user interaction experience is not very satisfied in the pilot project, which is due largely to the latency disparity between live messages and the broadcast content, as we have quantified through further measurements.

6.2.2 Further Enhancement on the Broadcaster Side

Joint optimization of servers and clients has been commonly employed in state-of-the-art streaming services to provide smooth streaming experience for heterogeneous viewers[71]. For crowdsourced livecast systems, it is necessary to include the massive sources in the optimization loop, which however can be quite challenging given their strong dynamics. Yet

the crowdsourced nature provides opportunities, too. We suggest that, through analyzing the enormous amount of historical activities of the broadcasters and viewers, the service provider may predict their behaviors in advance and accordingly improve the streaming quality.

Our measurement indicates that the adaptation strategy in Twitch is mainly based on CBR video. As such, good smoothness and low latency can hardly be both offered with limited bandwidth. For the viewer side, existing work [57] proposed a trade-off solution to distribute adaptive streaming in Twitch, which allows the heterogeneous viewer's devices to adaptively select the best-fit streaming quality. For the source side, we are working on a crowd uploading strategy that attempts to leverage the aggregated bandwidth of the many sources for speedy uploading. The live messaging and the associated social connections can play useful roles in the uploading, too.

6.2.3 Further Works on Mobile Gamecasing

Many mobile games are non-deterministic, e.g., a gamer touches and attacks the enemies that appear in different locations randomly. If a new location never appears in the collected touch data, the association rules cannot provide any feedback to the tile-based optimization module. To address this issue, we propose the following solution: after learning the rules from the original setting of tiles (e.g., the 5x5 setting in this paper), the association learning algorithm also adjusts the setting of tiles to learn the rules, e.g., from 5x5 to 4x4. For example, when an enemy appears at the location of tile #1 in the 5x5 setting, the gamer touches and attacks it. If there does not exist any rule about tile #1 in this setting, the prediction module searches the rules in the 4x4 setting. Because tile #1 in the 4x4 setting contains the contents of tiles #1, #2, #6, and #7 in the 5x5 setting, if the prediction module finds rules in the 4x4 setting, we consider these four tiles together to optimize the gamecasting.

If a game involves mostly randomly generated objects, the benefit would diminish. For example, in a Pinball game, the ball is only controlled by the player when launching it at the first step or redirecting it by two flippers. Then, the movements of this ball totally depend on the design of the playfield. In this scenario, the viewers' gazing patterns can hardly be predicted.

Bibliography

[1] V.K. Adhikari, Yang Guo, Fang Hao, M. Varvello, V. Hilt, M. Steiner, and Zhi-Li Zhang. Unreeling Netflix: Understanding and improving multi-cdn movie delivery. In *IEEE INFOCOM*, 2012.

[2] Hamed Ahmadi, Saman Zad Tootaghaj, Mahmoud Reza Hashemi, and Shervin Shirmohammadi. A game attention model for efficient bit rate allocation in cloud gaming. *Multimedia Systems*, 20(5):485–501, October 2014.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., 1993.

[4] Travis Andelin, Vasu Chetty, Devon Harbaugh, Sean Warnick, and Daniel Zappala. Quality selection for dynamic adaptive streaming over http with scalable video coding. In *ACM MMSys*, 2012.

[5] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: Ordering points to identify the clustering structure. *SIGMOD Rec.*, 28(2):49–60, 1999.

[6] Ramon Aparicio-Pardo, Karine Pires, Alberto Blanc, and Gwendal Simon. Transcoding live adaptive video streams at a massive scale in the cloud. In *ACM MMSys*, 2015.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[8] James H Bray and Scott E Maxwell. *Multivariate analysis of variance*, volume 54. Sage University Paper Series on Qualitative Research Methods, 1985.

[9] Zoya Bylinskii, Tilke Judd, Aude Oliva, Antonio Torralba, and Frédo Durand. What do different evaluation metrics tell us about saliency models? *arXiv preprint arXiv:1604.03605*, 2016.

[10] Fei Chen, Cong Zhang, Feng Wang, Jiangchuan Liu, Xiaofeng Wang, and Yuan Liu. Cloud-assisted live streaming for crowdsourced multimedia content. *IEEE Transactions on Multimedia*, 17(9):1471–1483, 2015.

[11] Li Chen, Baochun Li, and Bo Li. Surviving failures with performance-centric bandwidth allocation in private datacenters. In *IEEE IC2E*, 2016.

[12] Yimin Chen, Jingchao Sun, Rui Zhang, and Yanchao Zhang. Your song your way: Rhythm-based two-factor authentication for multi-touch mobile devices. In *IEEE INFOCOM*, 2015.

[13] Xu Cheng, Jiangchuan Liu, and C. Dale. Understanding the characteristics of Internet short video sharing: A youtube-based measurement study. *Multimedia, IEEE Transactions on*, 15(5):1184–1194, August 2013.

[14] Wei-Ta Chu and Yung-Chieh Chou. On broadcasted game video analysis: Event detection, highlight detection, and highlight forecast. *Multimedia Tools and Applications*, 76(7):9735–9758, April 2017.

[15] C. Cotta and J. M. Troya. *A Hybrid Genetic Algorithm for the 0–1 Multiple Knapsack Problem*, pages 250–254. Springer Vienna, Vienna, 1998.

[16] Khaled Diab and Mohamed Hefeeda. MASH: A rate adaptation algorithm for multiview video streaming over http. In *IEEE INFOCOM*, 2017.

[17] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(4):677–691, April 2017.

[18] A. Drachen, M. Yancey, J. Maguire, D. Chu, I. Y. Wang, T. Mahlmann, M. Schubert, and D. Klabajan. Skill-based differences in spatio-temporal team behaviour in defence of the ancients 2 (dota 2). In *IEEE GEM*, 2014.

[19] A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, 40(1):1–8, January 1988.

[20] Andrew Duchowski. *Eye Tracking Methodology: Theory and Practice*, volume 373. Springer Science & Business Media, 2007.

[21] Ali El Essaili, Zibin Wang, Eckehard Steinbach, and Liang Zhou. Qoe-based cross-layer optimization for uplink video transmission. *ACM Trans. Multimedia Comput. Commun. Appl.*, 12(1):2:1–2:22, August 2015.

[22] Timur Friedman, Ramon Caceres, and Alan Clark. RTP control protocol extended reports (RTCP XR). Technical report, 2003.

[23] Guanyu Gao, Han Hu, Yonggang Wen, and Cedric Westphal. Resource provisioning and profit maximization for transcoding in clouds: A two-timescale approach. *IEEE Transactions on Multimedia*, 19(4):836–848, April 2017.

[24] Mohammad Hajjat, Ruiqi Liu, Yiyang Chang, TS Eugene Ng, and Sanjay Rao. Application-specific configuration selection in the cloud: Impact of provider policy and potential of systematic testing. In *IEEE INFOCOM*, 2015.

[25] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[26] Qiyun He, Jiangchuan Liu, Chonggang Wang, and Bo Li. Coping with heterogeneous video contributors and viewers in crowdsourced live streaming: A cloud-based approach. *IEEE Transactions on Multimedia*, 18(5):916–928, May 2016.

[27] Qiyun He, Cong Zhang, and Jiangchuan Liu. Crowdtranscoding: Online video transcoding with massive viewers. *IEEE Transactions on Multimedia*, 19(6):1365–1375, June 2017.

[28] Mohamed Hefeeda, Ahsan Habib, Dongyan Xu, Bharat Bhargava, and Boyan Botev. Collectcast: A peer-to-peer service for media streaming. *Multimedia Systems*, 11(1):68–81, November 2005.

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[30] Todd Hoff. Justin.tv's live video broadcasting architecture, March 2010.

[31] Han Hu, Yonggang Wen, Tat-Seng Chua, Jian Huang, Wenwu Zhu, and Xuelong Li. Joint content replication and request routing for social video distribution over cloud cdn: A community clustering method. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(7):1320–1333, July 2016.

[32] Wenjie Hu and Guohong Cao. Energy optimization through traffic aggregation in wireless networks. In *IEEE INFOCOM*, 2014.

[33] Kyung-Wook Hwang, V. Gopalakrishnan, R. Jana, Seungjoon Lee, V. Misra, K.K. Ramakrishnan, and D. Rubenstein. Joint-family: Enabling adaptive bitrate streaming in peer-to-peer video-on-demand. In *IEEE ICNP*, 2013.

[34] Van Jacobson, Ron Frederick, Steve Casner, and H Schulzrinne. RTP: A transport protocol for real-time applications. 2003.

[35] Adele Lu Jia, Siqi Shen, Dick H. J. Epema, and Alexandru Iosup. When game becomes life: The creators and spectators of online game replays and live streaming. *ACM Trans. Multimedia Comput. Commun. Appl.*, 12(4):47:1–47:24, August 2016.

[36] Junchen Jiang, Vyas Sekar, Henry Milner, Davis Shepherd, Ion Stoica, and Hui Zhang. CFA: A practical prediction system for video qoe optimization. In *USENIX NSDI*, 2016.

[37] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *USENIX NSDI*, 2017.

[38] Mehdi Kaytoue, Arlei Silva, Loïc Cerf, Wagner Meira, Jr., and Chedy Raïssi. Watch me playing, i am a professional: A first study on video game live streaming. In *ACM WWW*, 2012.

[39] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Introduction to NP-Completeness of Knapsack Problems*. Springer, 2004.

[40] F.P. Kelly, A.K. Maulloo, and D.K.H. Tan. Rate control for communication networks: Shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49(3):237–252, 1998.

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[42] Jean Le Feuvre and Cyril Concolato. Tiled-based adaptive streaming using MPEG-DASH. In *ACM MMSys*, 2016.

[43] Baochun Li, Zhi Wang, Jiangchuan Liu, and Wenwu Zhu. Two decades of Internet video streaming: A retrospective view. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1s):33:1–33:20, October 2013.

[44] Haitao Li, Xu Cheng, and Jiangchuan Liu. Understanding video sharing propagation in social networks: Measurement and analysis. *ACM Trans. Multimedia Comput. Commun. Appl.*, 10(4):33:1–33:20, July 2014.

[45] Zhenhua Li, Yan Huang, Gang Liu, Fuchen Wang, Zhi-Li Zhang, and Yafei Dai. Cloud transcoder: Bridging the format and resolution gap between Internet videos and mobile devices. In *ACM NOSSDAV*, 2012.

[46] Zhenyu Li, Jiali Lin, Marc-Ismael Akodjenou, Gaogang Xie, Mohamed Ali Kaafar, Yun Jin, and Gang Peng. Watching videos from everywhere: A study of the pptv mobile vod system. In *ACM IMC*, 2012.

[47] Zhicheng Li, Shiyin Qin, and Laurent Itti. Visual attention guided bit allocation in video compression. *Image and Vision Computing*, 29(1):1 – 14, January 2011.

[48] Xiaofei Liao, Hai Jin, Yunhao Liu, Lionel M. Ni, and Dafu Deng. Anysee: Peer-to-peer live streaming. In *IEEE INFOCOM*, 2006.

[49] Zimu Liu, Chuan Wu, Baochun Li, and Shuqiao Zhao. *Why Are Peers Less Stable in Unpopular P2P Streaming Channels?*, pages 274–286. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[50] Yonggang Wen Weizhan Zhang Qinghua Zheng Lu, Zongqing and Guohong Cao. Towards information diffusion in mobile social networks. *IEEE Transactions on Mobile Computing*, 15(5):1292–1304, May 2016.

[51] He Ma, Beomjoo Seo, and Roger Zimmermann. Dynamic scheduling on video transcoding for MPEG DASH in the cloud environment. In *ACM MMSys*, 2014.

[52] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, September 2010.

[53] Ricky K. P. Mok, Xiapu Luo, Edmond W. W. Chan, and Rocky K. C. Chang. QDASH: a qoe-aware DASH system. In *ACM MMSys*, 2012.

[54] Kianoosh Mokhtarian and Mohamed Hefeeda. Analysis of peer-assisted video-on-demand systems with scalable video streams. In *ACM MMSys*, 2010.

[55] Omar A. Niamut, Emmanuel Thomas, Lucia D'Acunto, Cyril Concolato, Franck Denoual, and Seong Yong Lim. MPEG DASH SRD: Spatial relationship description. In *ACM MMSys*, 2016.

[56] Yipei Niu, Bin Luo, Fangming Liu, Jiangchuan Liu, and Bo Li. When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. In *IEEE INFOCOM*, 2015.

[57] Karine Pires and Gwendal Simon. DASH in Twitch: Adaptive bitrate streaming in live game streaming platforms. In *ACM VideoNext*, 2014.

[58] Karine Pires and Gwendal Simon. Youtube live and twitch: A tour of user-generated live streaming systems. In *ACM MMSys*, 2015.

[59] Swaminathan Vasanth Rajaraman, Matti Siekkinen, Vesa Virkki, and Johan Torsner. Bundling frames to save energy while streaming video from lte mobile device. In *ACM MobiArch*, 2013.

[60] Jihoon Ryoo, Kiwon Yun, Dimitris Samaras, Samir R. Das, and Gregory Zelinsky. Design and evaluation of a foveated video streaming service for commodity client devices. In *ACM MMSys*, 2016.

[61] Henning Schulzrinne. Real time streaming protocol (RTSP). 1998.

[62] Ryan Shea, Di Fu, and Jiangchuan Liu. Towards bridging online game playing and live broadcasting: Design and optimization. In *ACM NOSSDAV*, 2015.

[63] Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, and Mahadev Satyanarayanan. Scalable crowd-sourcing of video from mobile devices. In *ACM MobiSys*, 2013.

[64] I Sodagar. The MPEG-DASH standard for multimedia streaming over the Internet. *IEEE MultiMedia*, 18(4):62–67, April 2011.

[65] Yale Song. Real-time video highlights for yahoo esports. *CoRR*, abs/1611.08780, 2016.

[66] Yu Sun, I. Ahmad, Dongdong Li, and Ya-Qin Zhang. Region-based rate control and bit allocation for wireless video transmission. *IEEE Transactions on Multimedia*, February 2006.

[67] Christian Timmerer. Dynamic Adaptive Streaming over HTTP (DASH): Past, present, and future. `https://goo.gl/6oxenp`, November 2013.

[68] Maria Torres Vega, Decebal Constantin Mocanu, and Antonio Liotta. Unsupervised deep learning for real-time assessment of video streaming services. *Multimedia Tools and Applications*, 76(21):22303–22327, November 2017.

[69] Bolun Wang, Xinyi Zhang, Gang Wang, Haitao Zheng, and Ben Y. Zhao. Anatomy of a personalized livestreaming system. In *ACM IMC*, 2016.

[70] Feng Wang, Jiangchuan Liu, Minghua Chen, and Haiyang Wang. Migration towards cloud-assisted live media streaming. *IEEE/ACM Transactions on Networking*, 24(1):272–282, February 2016.

[71] Zhi Wang, Lifeng Sun, Chuan Wu, Wenwu Zhu, and Shiqiang Yang. Joint online transcoding and geo-distributed delivery for dynamic adaptive streaming. In *IEEE INFOCOM*, 2014.

[72] Stefan Wilk, Roger Zimmermann, and Wolfgang Effelsberg. Leveraging transitions for the upload of user-generated mobile video. In *ACM MoVid*, 2016.

[73] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, January 2008.

[74] Yu Wu, Chuan Wu, Bo Li, and Francis C.M. Lau. vskyconf: Cloud-assisted multi-party mobile video conferencing. In *ACM SIGCOMM Workshop on MCC*, 2013.

[75] Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. iAware: Making live migration of virtual machines interference-aware in the cloud. *IEEE Transactions on Computers*, 63(12):3012–3025, December 2014.

[76] Pu Yang, Brent E Harrison, and David L Roberts. Identifying patterns in combat that are predictive of success in moba games. In *FDG*, 2014.

[77] Fang Yu, Qian Zhang, Wenwu Zhu, and Ya-Qin Zhang. Qos-adaptive proxy caching for multimedia streaming over the Internet. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(3):257–269, March 2003.

[78] Cong Zhang, Qiyun He, Jiangchuan Liu, and Zhi Wang. Exploring viewer gazing patterns for touch-based mobile gamecasting. *IEEE Transactions on Multimedia*, 19(10):2333–2344, October 2017.

[79] Cong Zhang and Jiangchuan Liu. On crowdsourced interactive live streaming: A twitch.tv-based measurement study. In *ACM NOSSDAV*, 2015.

[80] Cong Zhang, Jiangchuan Liu, Ming Ma, Lifeng Sun, and Bo Li. Seeker: Topic-aware viewing pattern prediction in crowdsourced interactive live streaming. In *ACM NOSSDAV*, 2017.

[81] Cong Zhang, Jiangchuan Liu, and Haiyang Wang. Towards hybrid cloud-assisted crowdsourced live streaming: Measurement and analysis. In *ACM NOSSDAV*, 2016.

[82] Lei Zhang, Feng Wang, and Jiangchuan Liu. Mobile instant video clip sharing: Modeling and enhancing view experience. In *IEEE IWQoS*, 2016.

[83] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Y. S. P. Yum. Coolstreaming/donet: a data-driven overlay network for peer-to-peer live media streaming. In *IEEE INFOCOM*, 2005.

[84] Chao Zhou, Xinggong Zhang, Longshe Huo, and Zongming Guo. A control-theoretic approach to rate adaptation for dynamic http streaming. In *IEEE VCIP*, 2012.

[85] Fen Zhou, Shakeel Ahmad, Eliya Buyukkaya, Raouf Hamzaoui, and Gwendal Simon. Minimizing server throughput for low-delay live streaming in content delivery networks. In *ACM NOSSDAV*, 2012.

[86] Yipeng Zhou, Liang Chen, Mi Jing, Shenglong Zou, and Richard Tianbai Ma. Design, implementation, and measurement of a crowdsourcing-based content distribution platform. *ACM Trans. Multimedia Comput. Commun. Appl.*, 12(5s):80:1–80:23, November 2016.

[87] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. Characteristics of youtube network traffic at a campus network - measurements, models, and implications. *Comput. Netw.*, 53(4):501–514, March 2009.