Cloud-assisted Crowdsourced Livecast

CONG ZHANG AND JIANGCHUAN LIU, Simon Fraser University
HAIYANG WANG, University of Minnesota Duluth

The recent two years have witnessed an explosion of a new generation of livecast services, represented by *Twitch.tv*, *GamingLive*, and *Dailymotion*, to name but a few. With such a livecast service, geo-distributed Internet users can broadcast any event in real-time, e.g., game, cooking, drawing, etc., to viewers of interest. Its crowdsourced nature enables rich interactions among broadcasters and viewers, but also introduces great challenges to accommodate their great scales and dynamics. To fulfill the demands from a large number of heterogeneous broadcasters and geo-distributed viewers, expensive server clusters have been deployed to ingest and transcode live streams. Yet our Twitch-based measurement shows that a significant portion of the unpopular and dynamic broadcasters are consuming considerable system resources; in particular, 25% of bandwidth resources and 30% of computational capacity are used by the broadcasters who do not have any viewers at all. In this article, 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.

CCS Concepts: • **Information systems** → **Multimedia streaming**;

Additional Key Words and Phrases: Crowdsourced livecast, public clouds, workload migration, resource allocation

ACM Reference format:
Cong Zhang and Jiangchuan Liu and Haiyang Wang. 2010. Cloud-assisted Crowdsourced Livecast. *ACM Trans. Multimedia Comput. Commun. Appl.* 9, 4, Article 39 (March 2010), 23 pages.
DOI: 0000001.0000001

1 INTRODUCTION

Empowered by the high-performance personal devices (e.g., desktop PCs and mobile phones) and high-speed communication networks (e.g., optical networks and LTE), the paradigm of

This publication was made possible by NPRP grant #[8-519-1-108] from the Qatar National Research Fund (a member of Qatar Foundation). The findings achieved herein are solely the responsibility of the authors. Dr. J. Liu's work is also supported by the Natural Sciences and Engineering Research Natural Sciences and Engineering Research Council (NSERC) of Canada. H. Wang's work was supported by Chancellor's Small Grant and Grant-in-aid programs from the University of Minnesota. Author's addresses: C. Zhang and J. Liu, School of Computing Science, Simon Fraser University, Canada; H. Wang, Department of Computer Science, University of Minnesota Duluth, USA..

live streaming service has shifted from the conventional single source, to multi-source, to many sources, and now toward *crowdsource* [17], where the available media sources for the content of interest become highly diverse and scalable. In recent years, crowdsourced livecast has emerged as a powerful and popular streaming service over the Internet [9]. Such livecast services as *Twitch.tv*[1] (or Twitch for short), *GamingLive*[2], and *Dailymotion*[3], allow Internet users to broadcast cooking shows, costume design, music-making, game playthrough[4], etc., attracting an increasing number of viewers around the world. One recent report from Twitch[5] 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[6][7]. Currently, it has 31 service regions (i.e., ingesting regions) across five continents[8].

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 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 [15][18]. 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[9]. We, therefore, explore the feasibility of using dedicated servers and public clouds cooperatively. Through a series of measurements based on Amazon EC2[10] (EC2 for short) and PlanetLab[11] 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 article, 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

[1]www.twitch.tv, owned by Amazon.com in September, 2014.
[2]www.gaminglive.tv
[3]www.dailymotion.com
[4]When game players play games, they also broadcast the monitor contents from their game devices to fellow viewers with the real-time comments.
[5]https://blog.twitch.tv/twitch-engineering-an-introduction-and-overview-a23917b71a25#.87r28fv6u
[6]https://dotesports.com/general/twitch-paris-data-center-europe-174
[7]http://www.techradar.com/news/gaming/twitch-flips-the-switch-on-a-new-australian-data-centre-1307999
[8]https://twitchstatus.com/
[9]http://highscalability.com/blog/2010/3/16/justintvs-live-video-broadcasting-architecture.html
[10]https://aws.amazon.com/ec2
[11]https://www.planet-lab.org/

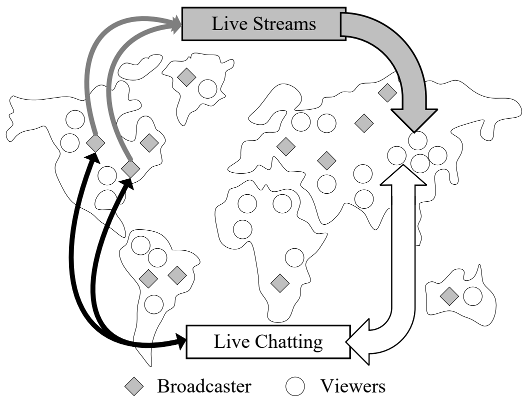


Fig. 1. A generic system diagram of crowdsourced livecast platforms

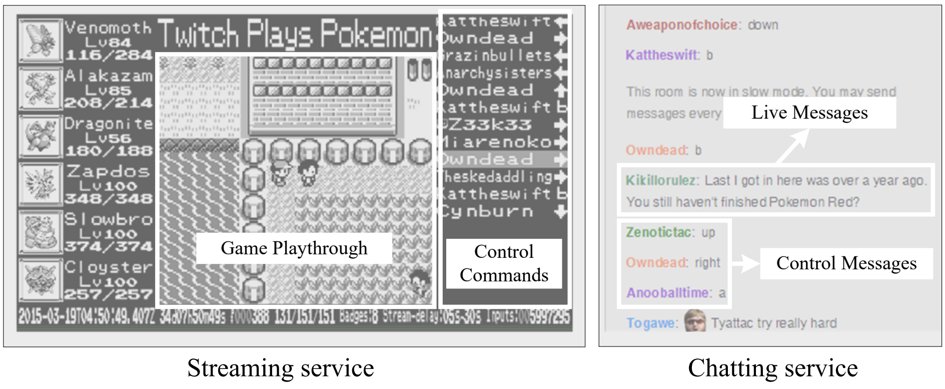
problems in ingesting and transcoding steps, considering the diverse capacities and expenses in different regions. Trace-driven evaluations show that our proposed solutions migrate up to 59.9% of workloads from the dedicated servers to the public cloud and reduce about 20% of leasing cost compared with other cloud-assisted strategies. The remainder of this article is organized as follows. We introduce the background of crowdsourced livecast systems in Section 2. We present the Twitch-based measurement and emphasize the challenge of handling unpopular broadcasters in Section 3. We introduce the EC2-based measurement and propose the CACL framework in Section 4. We formulate and solve the resource allocation problem in Section 5. The trace-driven simulations evaluate the performance of our design in Section 6. After the discussion of related work in Section 7, we conclude our work and further offer some potential research directions in Section 8.

2 BACKGROUND

In this section, we briefly introduce the system diagram of crowdsourced livecast platforms. As shown in Figure 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)[12], to maintain the low-latency communication. Then, the streams are transcoded to multi-quality formats, e.g., HTTP Live Streaming (HLS)[13], 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.

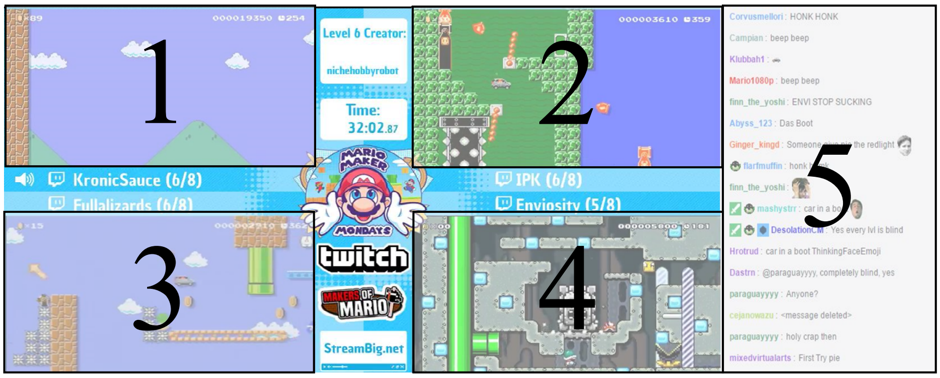
For example, the crowdsourced game stream "TwitchPlaysPokemon"[14], as shown in Figure 2a, offered the live stream and emulator for the game "Pokemon Red[15]", in which players (also as the viewers in Twitch) simultaneously send the control messages of Pokemon through the IRC (Internet Relay Chat) protocol and live messages in Twitch. That said, the viewers are no longer passive, but can affect the progress of the broadcast as well.

[12]https://en.wikipedia.org/wiki/Real-Time_Messaging_Protocol
[13]https://en.wikipedia.org/wiki/HTTP_Live_Streaming
[14]https://en.wikipedia.org/wiki/Twitch_Plays_Pokemon
[15]Pokemon Red is a role-playing video game.



Aweaponofchoice: down
Kattheswift: b
This room is now in slow mode. You may send messages every
Owndead: b
Kikillorulez: Last I got in here was over a year ago. You still haven't finished Pokemon Red?
Zenotictac: up
Owndead: right
Anooballtime: a
Togawe: Tyattac try really hard

(a) TwitchPlaysPokemon



Corvusmellori : HONK HONK
Campian : beep beep
Klubbah1 :
Mario1080p : beep beep
finn_the_yoshi : ENVI STOP SUCKING
Abyss_123 : Das Boot
Ginger_kingd : Someone give pie the redlight
flarfmuffin : honk honk
finn_the_yoshi :
mashystrr : car in a boot
DesolationCM : Yes every lvl is blind
Hrotrud : car in a boot ThinkingFaceEmoji
Dastrn : @paraguayyyy, completely blind, yes
paraguayyyy : Anyone?
cejanowazu : <message deleted>
paraguayyyy : holy crap then
mixedvirtualarts : First Try pie

(b) Super Mario Maker Competition

Fig. 2. The illustrations of two crowdsourced livecast streams

This truly crowdsourced game streaming attracted more than 1.6 million players and 55 million viewers. Figure 2b demonstrates another typical livecast example, in which four broadcasters are playing Super Mario Maker[16]. We split this streaming screenshot into five areas. Four players' games are rendered in area #1-#4, respectively. The viewers can discuss the broadcasters' game operations in the chatting window in area #5. 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[17]; yet many broadcasters have only one or two viewers, or even none.

[16]Super Mario Maker is the evolution version of the classic video game Super Mario Bro.
[17]https://www.twitch.tv/twitchplayspokemon

3 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 impacts 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[18].

3.1 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[19], our multi-threaded crawler[20] obtained information from each broadcaster and the official system dashboard[21]. We retrieved both the broadcaster dataset and stream dataset from it[22]. We have excluded certain outliers[23] 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[24] 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).

3.2 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[25] and OBS[26]). 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 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 4 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,

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

[19]http://dev.twitch.tv/

[20]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.

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

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

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

[24]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.

[25]https://www.xsplit.com/

[26]Open Broadcaster Software, https://obsproject.com/

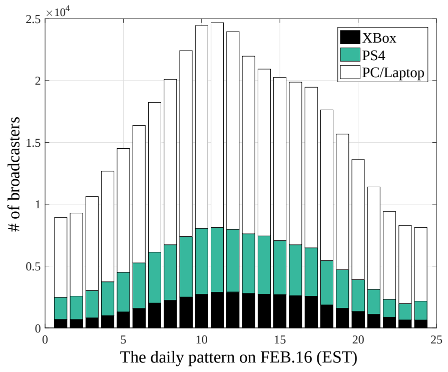


Fig. 3. The distribution of three types of devices (Eastern Standard Time)

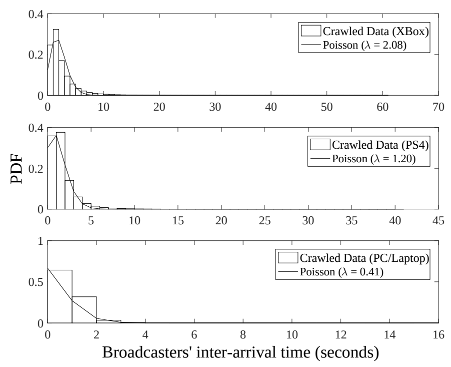


Fig. 4. Broadcasters' inter-arrival time

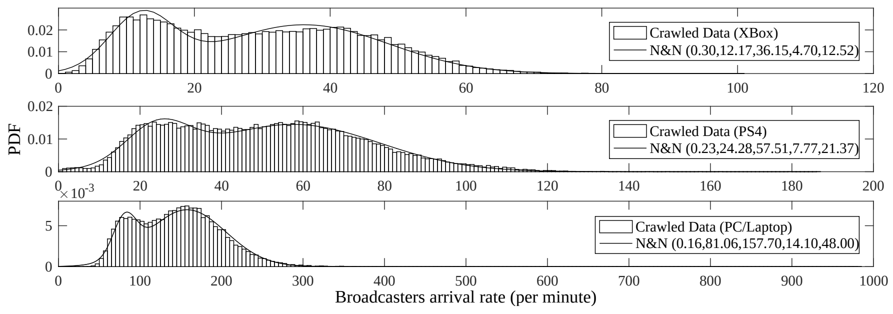
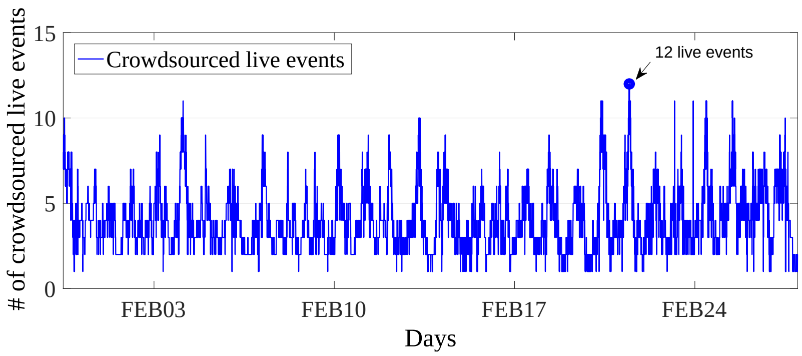


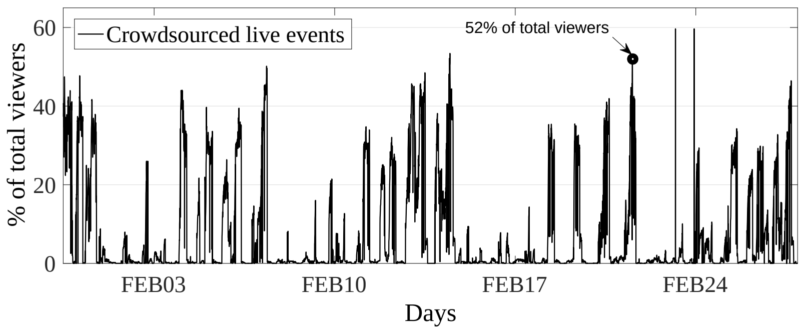
Fig. 5. Broadcasters' arrival rate

respectively. Note that a Poisson Distribution can be used to fit the inter-arrival time of three platforms with different parameters λ.

Figure 5 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 4 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 5, 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.



(a) Daily patterns



(b) Effects of crowdsourced live events

Fig. 6. Characteristics of crowdsourced live events

3.3 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 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 6a plots the number of live events during one month[27]. 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 6b. 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 5.

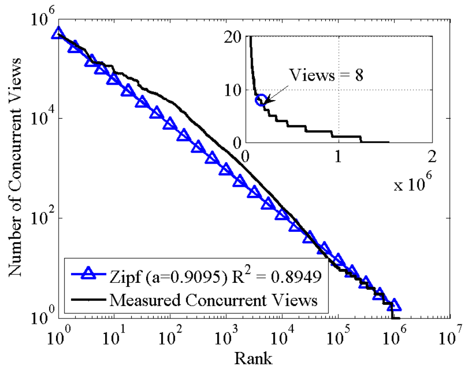


Fig. 7. Broadcasters rank ordered by popularity

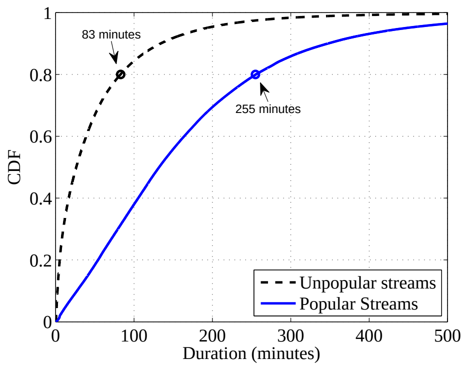


Fig. 8. The distribution of duration

3.4 Popularity of Crowdsourced Live Broadcasters

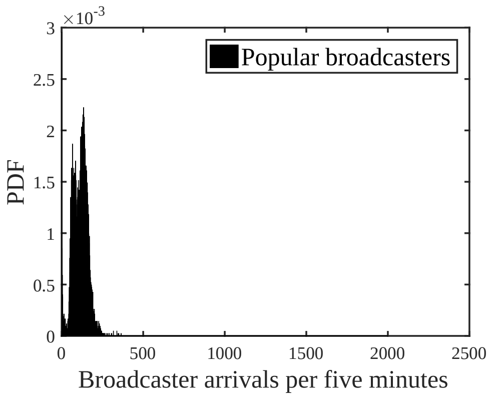
We then focus on the distribution of broadcaster's popularity, which is a key feature in previous studies for multimedia systems [4][13], and is also critical to answer our first question. We plot the highest number of concurrent viewers against the rank of the broadcasters (in terms of the popularity) in log-log scale in Figure 7. From this figure, we observe that the popularity of those broadcasters well exhibits a Zipf's pattern[28]. 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 7) 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.

3.5 Dynamics of Crowdsourced Live Broadcasters

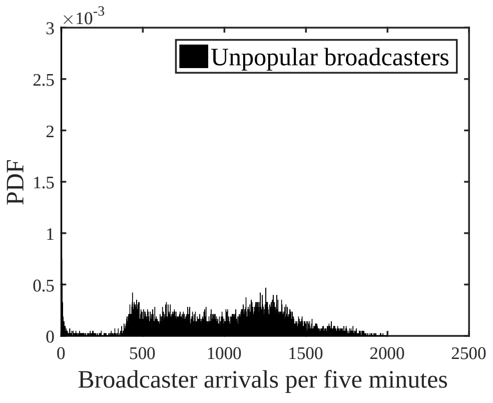
In the stream dataset, the unpopular streams account for 89.5% of all streams. We next try to answer two critical questions: (1) *How long are these unpopular streams?* (2) *Is there any difference between popular and unpopular streams in terms of live duration?* We compare the distribution of their duration with the popular streams, as shown in Figure 8. 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 9. 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 10a and 10b. Figure 10a illustrates that broadcaster

[27]Due to the space limitation, only four date labels are displayed.
[28]We use the coefficient of determination, denoted R^2, to illustrate how well our measured data fit the Zipf's law.

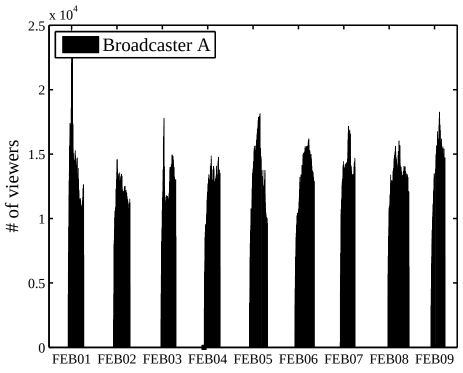


(a) Popular broadcasters

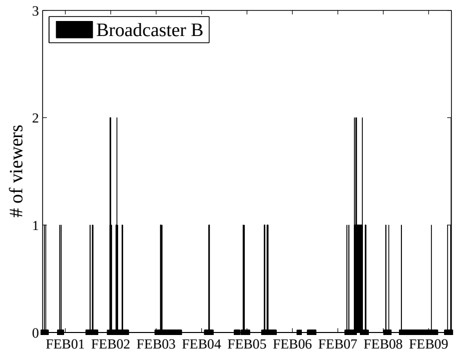


(b) Unpopular broadcasters

Fig. 9. Broadcaster arrivals per five minutes



(a) Popular broadcaster A



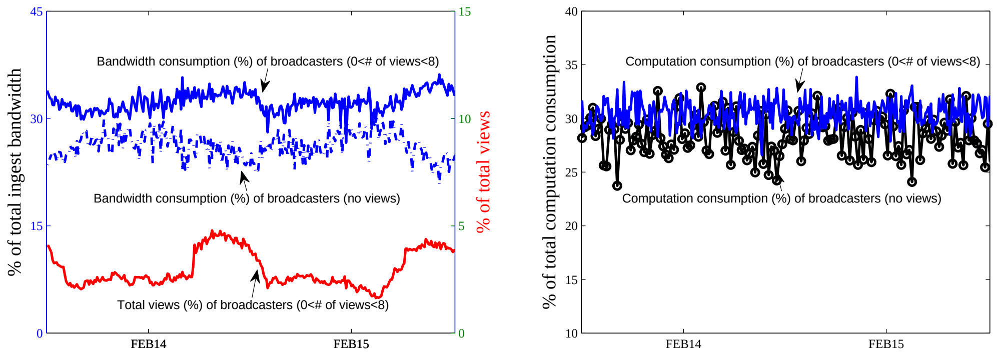
(b) Unpopular broadcaster B

Fig. 10. Two examples in the broadcaster dataset

A has a regular schedule with a stable live duration, attracting a large number of viewers. Figure 10b 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.

3.6 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 [2], which provides the empirical CPU cycles measurements under different transcoding settings. Figure 11 shows the proportion of bandwidth/computational consumption of two types of



(a) Bandwidth consumption (b) Computational consumption

Fig. 11. Different types of broadcasters' resource consumptions

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.

4 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 impacts of current dynamic, unpredictable, and irregular workloads cost-effectively.

4.1 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 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

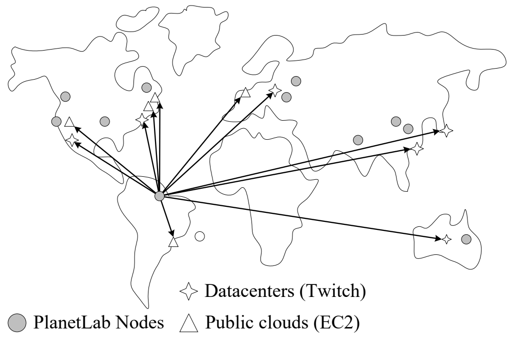


Fig. 12. The diagram of EC2-based measurement

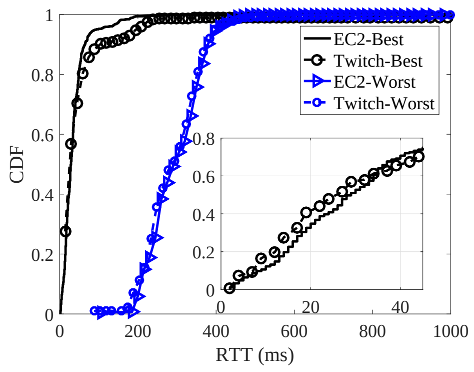


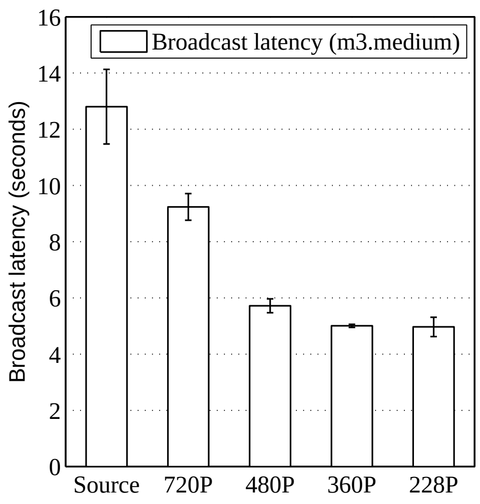
Fig. 13. RTT comparison

with Ubuntu 14.04[29] and Nginx-RTMP module[30]. 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 12. We measure the RTTs between 224 PlanetLab nodes and 26 ingesting servers (18 in Twitch[31] and 8 on EC2) and acquire the following results.

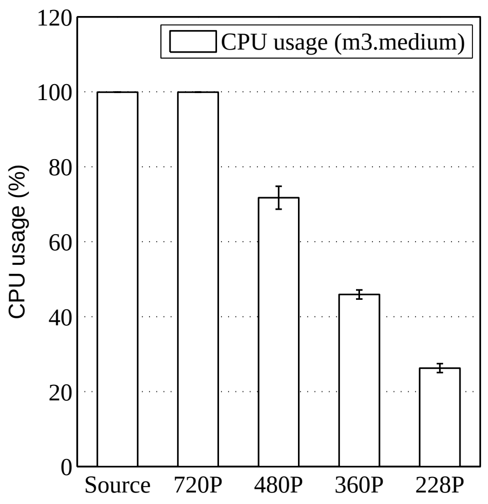
4.1.1 Round-trip Time. Figure 13 shows the RTT comparison for the Twitch and EC2 cases. We can observe that about 60% of broadcasters in the Twitch-best case have a quite low RTT (less than 35 ms), but the disparity between the two best cases is quite small, as shown in the small figure in Figure 13. 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.

4.1.2 Broadcast Latency. Our previous work defined two types of latencies in crowdsourced livecast platforms [21]: (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.

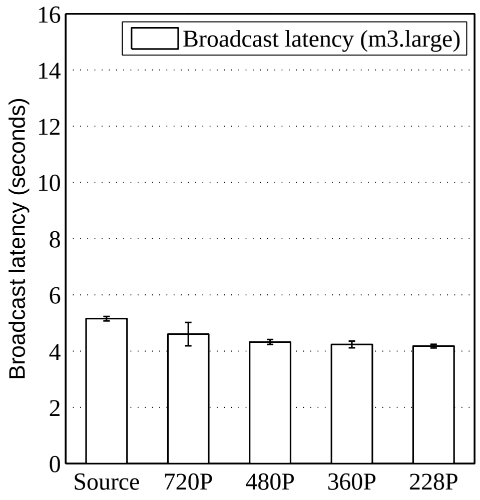
[29]http://releases.ubuntu.com/14.04/
[30]https://github.com/arut/nginx-rtmp-module
[31]Twitch had only 18 ingesting regions during our study.



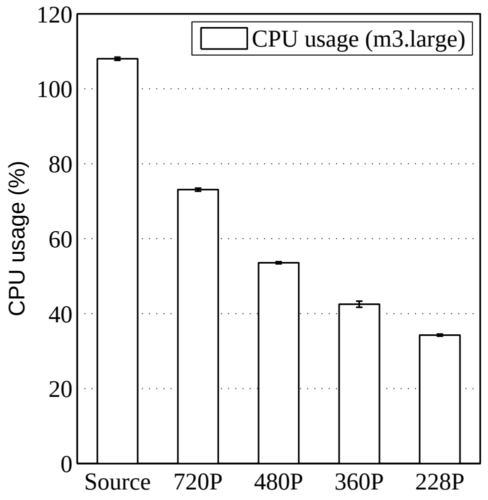
(a) Broadcast latency(m3.m)



(b) CPU usage (m3.m)



(c) Broadcast latency (m3.l)



(d) CPU usage (m3.l)

Fig. 14. Broadcast latency in different instances

To measure the broadcast latency on public clouds, we lease two types of instances (m3.medium and m3.large[32]) from Amazon's Oregon data-center. We deploy a PC (Dell

[32]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

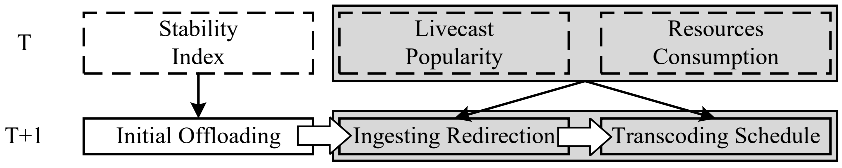


Fig. 15. The framework of Cloud-assisted Crowdsourced Livecast (CACL)

7010) with OBS as the broadcaster's device and a laptop (Samsung NP355V5C) with VLC[33] 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 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 14 with the average values and standard deviations. Figure 14a and 14b 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[21]. Because another m3.large instance has two vCPUs[34], it has better performance, as shown in Figure 14c and 14d. From Figure 14c, we observe that all broadcast latencies of various settings are decreased to about 5 seconds with the sufficient computational capacity. From Figure 14d, 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.

4.2 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 15: (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 workload is offloaded to a certain ingesting region, the corresponding transcoding workload has to be processed in the same region to reduce the broadcast latency.

[33]VLC is a free and open source multimedia player and framework, http://www.videolan.org/vlc/index.html
[34]Each of vCPUs is a hyperthread of an Intel Xeon core

4.3 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 depends on the duration and schedule of a broadcaster's historical streams. 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.

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

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. We will evaluate the effectiveness of s-index in Section 6. In the next section, we first propose the problem formulations of Ingesting Redirection and Transcoding Schedule, and then propose the cost-effective solution through the heuristic algorithms.

5 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 [12][14][2]. 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.

5.1 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 [20][3][18]. We define $L_{(b,r)}^{(t)}$ as the broadcast latency if b selects ingesting area r. It can be calculated as:

$$L_{(b,r)}^{(t)} = l_{(b,r)}^{(t)} + l_r^{(t)} + l_{(r,v)} \tag{2}$$

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:

$$U^{(t)}(b,r) = G^{(t)}(b,r) \cdot N_b^{(t)} \tag{3}$$

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 [11], we define $G^{(t)}(b,r)$ as follows:

$$G^{(t)}(b,r) = \alpha + ln(1 - \beta L_{(b,r)}^{(t)}) \tag{4}$$

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.

$$\underset{e \in E^{(t)}}{\text{Maximize}}\ F(A^{(t)}) = \min_{\substack{b \in e \\ r \in R}} \{U^{(t)}(b,r)\} \tag{5}$$

subject to:

Bandwidth Availability Constraint:

$$\forall r \in R, W_r^{(t)} \leq \mathbb{W}_r \tag{6}$$

Bandwidth Cost Constraint:

$$\sum_{r \in R} \frac{W_r^{(t)}}{\mathcal{W}} \cdot Cost_w(r) \cdot I(r) \leq K_w \tag{7}$$

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 (6) asks that at any given time, the bandwidth demands have to be satisfied. The total budget constraint (7) asks that at any given time, the total cost of leasing instances does not surplus total budget K_w.

5.2 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 $L^{(t)}_{(b,r)}$ in the equation (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.

$$L^{(t)}_{(b,r,v)} = l^{(t)}_{(b,r)} + l^{(t)}_{(q_b,q_v)} + l_{(r,v)} \tag{8}$$

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

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

$$U^{(t)}(b,r) = \sum_{v \in V} G^{(t)}(b,r,v) \cdot N^{(t)}_{(b,v)} \tag{9}$$

where $N^{(t)}_{(b,v)}$ 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:

$$\begin{aligned} G^{(t)}(b,r,v) &= \alpha + ln(1 - \beta L^{(t)}_{(b,r,v)}) \\ &= \alpha + ln(1 - \beta(l^{(t)}_{(b,r)} + l^{(t)}_{(q_b,q_v)} + l^{(t)}_{(r,v)})) \end{aligned} \tag{10}$$

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 [19].

Our objective is extended to a new version as:

$$\underset{e \in E^{(t)}}{\text{Maximize}}\ F(A^{(t)}) = \min_{\substack{b \in e \\ r \in R}} \{U^{(t)}(b,r)\} \tag{11}$$

subject to:
Previous Constraints: (6), (7)
Computational Availability Constraint:

$$\forall r \in R, C^{(t)}_r \leq \mathbb{C}_r \tag{12}$$

Computational Cost Constraint:

$$\sum_{r \in R} \frac{C^{(t)}_r}{\mathcal{C}} \cdot Cost_c(r) \cdot I(r) \leq K_c \tag{13}$$

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 (12) guarantees that at any given time t, the consumption of computational resources in each transcoding task can be satisfied. The budget constraint (13) guarantees that the computational cost is lower than the budget K_c, which we assume can at least serve all offloading workloads.

5.3 Solution

The objective function (11) has four constraints (6) (7) (12), and (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 [8]. Moreover, our measurement results in Section 4.1 also reveal that generating low-latency live streams will consume a vast amount of computational resources. If we relax constraints (6) and (7), other constraints can still work for the optimization objective function (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 [6].

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 1-11). We then remove most parts of the solution space (line 12-21) 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 (14) and find the area r^* in the equation (15), which is derived from [5]. 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 into available service areas.

$$u^{(t)}(b, r) = \frac{C_r^{(t)} \cdot U^{(t)}(b, r)}{c(b)} \tag{14}$$

$$r_b^* = \underset{r \in R}{\operatorname{argmin}} \; \{u^{(t)}(b, r)\} \tag{15}$$

6 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.

6.1 Efficiency of Resource Allocation

We first evaluate the performance of our resource allocation algorithm using the traces in the EC2-based measurement (Section 4.1). 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

Algorithm 1 ScaleDecrease()

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

Algorithm 2 ResourceAllocation()

```
1: for broadcaster's assignment $(b,r)$ in $A^{(t)}$ do
2:    Add $u^{(t)}(b,r)$ into $u_b^{(t)}$;
      // Calculate $u^{(t)}(b,r)$ using the equation (14)
3: end for
4: $B_{sorted} \leftarrow$ Sorted broadcasters in descendant order of $u^{(t)}(b,r_b^*)$;
   // Get $r_b^*$ from $u_b^{(t)}$ according to the equation (15);
5: for broadcaster $b \in B_{sorted}$ do
6:    $r_{sorted} \leftarrow$ Sorted available area $r$ of $b$ in descendant order of $u_b^{(t)}$;
7:    for region $r \in r_{sorted}$ do
8:       if $C_r^{(t)} - c(b) \geq 0$ then
9:          $A^{(t)} \leftarrow A^{(t)} - (b,\cdot)$;
            // Remove all assignment of $b$
10:         $C_r^{(t)} \leftarrow C_r^{(t)} + c(b)$;
11:         $A^{(t)} \leftarrow A^{(t)} + (b,r)$;
12:      end if
13:   end for
14: end for
15: return $A^{(t)}$
```

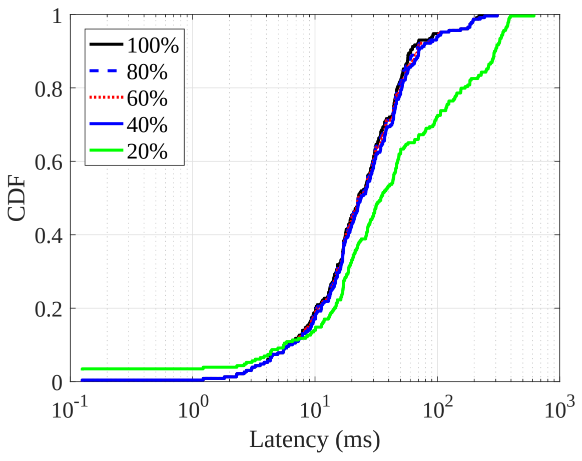


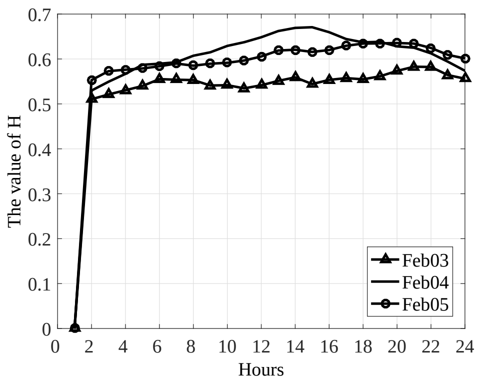
Fig. 16. The impacts of computational capacity

resolution of every broadcaster according to the Twitch-based measurement in Section 3. 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 [2]. To clearly demonstrate the effectiveness of our solution, we adjust the computational capacity from 100% to 20%. Figure 16 demonstrates the impacts 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.

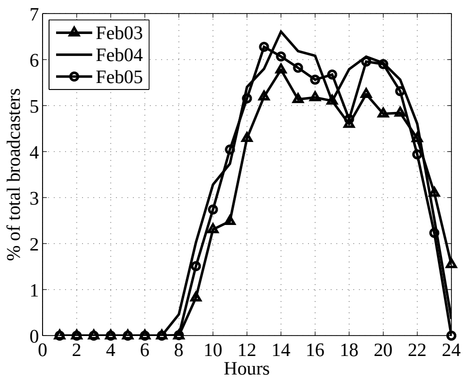
6.2 Trace-driven Simulation

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[35] in Twitch is from 10 to 40 seconds [21], 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 4.1. The algorithms are launched every five minutes, which also is the time slot of crawling data.

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

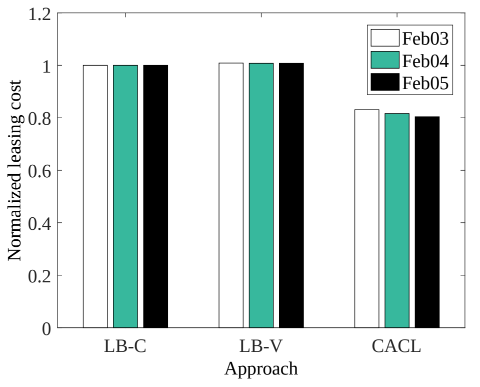


(a) The evolution of H

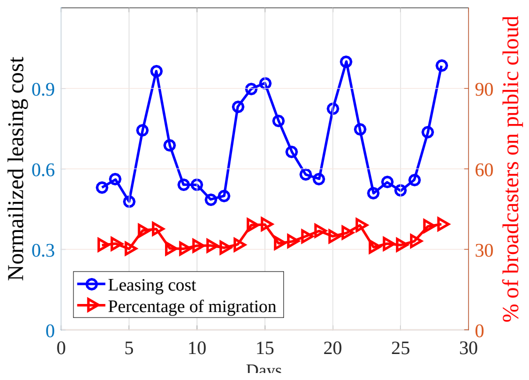


(b) Percentage of broadcasters (EC2)

Fig. 17. The impacts of threshold H



(a) Lease cost of three approaches



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

Fig. 18. The performance of proposed solutions

We first study the impacts 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 $H = 0$ 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 17 illustrates the evolution of H and its impacts for the public cloud during three days (Feb 3rd-5th, 2015). From Figure 17a, 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 impacts 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 18a 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 18b, 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.

7 RELATED WORK

Some recent studies have already focused on the crowdsourced livecast service. Kaytoue *et al.* [10] introduced the characteristics of Twitch from the perspective of web communities. To address the transcoding problem for non-professional broadcasters, Aparicio-Pardo *et al.* [2] first analyzed the Twitch dataset, and then proposed an optimal model to improve the viewer's satisfaction. Shea *et al.* [16] 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.* [7] 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 and irregular workloads in crowdsourced livecast platforms; second, our cloud-assisted design utilizes the flexible resources from public clouds as a complement, with minimum change to the existing architecture. This article is extended from our preliminary conference version [22] in three ways. First, we have analyzed the characteristics of Twitch broadcasters in different types of devices (i.e., PC/Laptop, PS4, and XBox) and measured Amazon EC2 instance for transcoding. These motivate our design of the cloud-assisted crowdsourced livecast. Second, we have added a basic model to explain the problem formulation step by step, which provides the details in our design. In addition, the extended trace-driven simulation exhibits the efficiency of our proposed algorithm based on the measurement results from Amazon EC2 and PlanetLab nodes.

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[36])" 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[37] supplies both the on-demand and live transcoding service to customers. Similar services also include Zencoder[38], PandaStream[39], EncoderCloud[40], 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.* [12] presented "Cloud Transcoder" to transcode the high resolution and heterogeneous videos from mobile devices. Ma *et al.* [14] 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 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.

8 CONCLUSION AND FUTURE WORK

In this article, 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.

We are currently conducting more simulations to evaluate and improve CACL with the datasets and traces from other cloud providers and crowdsourced livecast services. We expect to develop a prototype for further verification and evaluation. We are also interested in exploring other open issues such as designing better resource allocation mechanisms and extending CACL to multiple cloud platforms, e.g., Microsoft Azure.

REFERENCES

[1] V.K. Adhikari, Yang Guo, Fang Hao, M. Varvello, V. Hilt, M. Steiner, and Zhi-Li Zhang. 2012. Unreeling Netflix: Understanding and Improving Multi-CDN Movie Delivery. In *Proceedings of IEEE INFOCOM.*

[2] Ramon Aparicio-Pardo, Karine Pires, Alberto Blanc, and Gwendal Simon. 2015. Transcoding Live Adaptive Video Streams at a Massive Scale in the Cloud. In *Proceedings of ACM MMSys.*

[3] Li Chen, Baochun Li, and Bo Li. 2016. Surviving Failures with Performance-Centric Bandwidth Allocation in Private Datacenters. In *Proceedings of IEEE IC2E.*

[4] Xu Cheng, Jiangchuan Liu, and Cameron Dale. 2013. Understanding the Characteristics of Internet Short Video Sharing: A YouTube-Based Measurement Study. *IEEE Transactions on Multimedia* 15, 5 (Aug 2013), 1184–1194.

[36]http://aws.amazon.com/elastictranscoder/
[37]http://www.bitmovin.net/
[38]https://zencoder.com/en/
[39]https://www.pandastream.com/
[40]http://www.encodercloud.com/

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

[6] A. Drexl. 1988. A Simulated Annealing Approach to the Multiconstraint Zero-one Knapsack Problem. *Computing* 40, 1 (Jan 1988), 1–8.

[7] Ali El Essaili, Zibin Wang, Eckehard Steinbach, and Liang Zhou. 2015. QoE-Based Cross-Layer Optimization for Uplink Video Transmission. *ACM Trans. Multimedia Comput. Commun. Appl.* 12, 1 (Aug 2015), 2:1–2:22.

[8] Mohammad Hajjat, Ruiqi Liu, Yiyang Chang, TS Eugene Ng, and Sanjay Rao. 2015. Application-specific Configuration Selection in the Cloud: Impact of Provider Policy and Potential of Systematic Testing. In *Proceedings of IEEE INFOCOM.*

[9] Adele Lu Jia, Siqi Shen, Dick H. J. Epema, and Alexandru Iosup. 2016. When Game Becomes Life: The Creators and Spectators of Online Game Replays and Live Streaming. *ACM Trans. Multimedia Comput. Commun. Appl.* 12, 4 (Aug 2016), 47:1–47:24.

[10] Mehdi Kaytoue, Arlei Silva, Loïc Cerf, Wagner Meira, Jr., and Chedy Raïssi. 2012. Watch Me Playing, I Am a Professional: A First Study on Video Game Live Streaming. In *Proceedings of ACM WWW.*

[11] F.P. Kelly, A.K. Maulloo, and D.K.H. Tan. 1998. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *Journal of the Operational Research Society* 49, 3 (1998), 237–252.

[12] Zhenhua Li, Yan Huang, Gang Liu, Fuchen Wang, Zhi-Li Zhang, and Yafei Dai. 2012. Cloud Transcoder: Bridging the Format and Resolution Gap Between Internet Videos and Mobile Devices. In *Proceedings of ACM NOSSDAV.*

[13] Zimu Liu, Chuan Wu, Baochun Li, and Shuqiao Zhao. 2009. Why Are Peers Less Stable in Unpopular P2P Streaming Channels? In *NETWORKING 2009.* Lecture Notes in Computer Science, Vol. 5550. 274–286.

[14] He Ma, Beomjoo Seo, and Roger Zimmermann. 2014. Dynamic Scheduling on Video Transcoding for MPEG DASH in the Cloud Environment. In *Proceedings of ACM MMSys.*

[15] Yipei Niu, Bin Luo, Fangming Liu, Jiangchuan Liu, and Bo Li. 2015. When Hybrid Cloud Meets Flash Crowd: Towards Cost-Effective Service Provisioning. In *Proceedings of IEEE INFOCOM.*

[16] Ryan Shea, Di Fu, and Jiangchuan Liu. 2015. Towards Bridging Online Game Playing and Live Broadcasting: Design and Optimization. In *Proceedings of ACM NOSSDAV.*

[17] Pieter Simoens, Yu Xiao, Padmanabhan Pillai, Zhuo Chen, Kiryong Ha, and Mahadev Satyanarayanan. 2013. Scalable Crowd-sourcing of Video from Mobile Devices. In *Proceedings of ACM MobiSys.*

[18] Feng Wang, Jiangchuan Liu, Minghua Chen, and Haiyang Wang. 2016. Migration Towards Cloud-Assisted Live Media Streaming. *IEEE/ACM Transactions on Networking* 24, 1 (Feb 2016), 272–282.

[19] Yu Wu, Chuan Wu, Bo Li, and Francis C.M. Lau. 2013. vSkyConf: Cloud-assisted Multi-party Mobile Video Conferencing. In *Proceedings of ACM SIGCOMM Workshop on MCC.*

[20] Fei Xu, Fangming Liu, Linghui Liu, Hai Jin, Bo Li, and Baochun Li. 2014. iAware: Making Live Migration of Virtual Machines Interference-Aware in the Cloud. *IEEE Trans. Comput.* 63, 12 (Dec 2014), 3012–3025.

[21] Cong Zhang and Jiangchuan Liu. 2015. On Crowdsourced Interactive Live Streaming: A Twitch.Tv-based Measurement Study. In *Proceedings of ACM NOSSDAV.*

[22] Cong Zhang, Jiangchuan Liu, and Haiyang Wang. 2016. Towards Hybrid Cloud-assisted Crowdsourced Live Streaming: Measurement and Analysis. In *Proceedings of ACM NOSSDAV.*

Received February 2007; revised March 2009; accepted June 2009