MultiLive: Adaptive Bitrate Control for Low-Delay Multi-Party Interactive Live Streaming

Ziyi Wang, Yong Cui, *Member, IEEE*, Xiaoyu Hu, Xin Wang, *Member, IEEE*, Wei Tsang Ooi, *Member, IEEE*, Zhen Cao, and Yi Li

***Abstract*— In multi-party interactive live streaming, each user can act as both the sender and the receiver of a live video stream. Designing adaptive bitrate (ABR) algorithm for such applications poses three challenges: (i) due to the interaction requirement among the users, the playback buffer has to be kept small to reduce the end-to-end delay; (ii) the algorithm needs to decide what is the bitrate to receive and what is the set of bitrates to *send*; (iii) the delay and quality requirements between each pair of users may differ, for instance, depending on whether the pair is interacting directly with each other. To address these challenges, we first develop a quality of experience (QoE) model for multi-party live streaming applications. Based on this model, we design *MultiLive*, an adaptive bitrate control algorithm for the multi-party scenario. MultiLive models the many-to-many ABR selection problem as a non-linear programming problem. Solving the non-linear programming equation yields the target bitrate for each pair of sender-receiver. To alleviate system errors during the modeling and measurement process, we update the target bitrate through the buffer feedback adjustment. To address the throughput limitation of the uplink, we cluster the ideal streams into a few groups, and aggregate these streams through scalable video coding for transmissions. We also deploy the algorithm on a commercial live streaming platform that provides such services for more than 2300 users. The experimental results show that MultiLive outperforms the fixed bitrate algorithm, with 2-5× improvement in average QoE. Furthermore, the end-to-end delay is reduced to around 100 ms, much lower than the 400 ms threshold recommended for video conferencing.**

***Index Terms*— Multi-party interactive live streaming, adaptive bitrate control, available bandwidth measurement.**

Manuscript received March 23, 2020; revised April 23, 2021 and September 23, 2021; accepted November 14, 2021; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor J. Shin. Date of publication December 1, 2021; date of current version April 18, 2022. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800303, in part by the National Natural Science Foundation of China (NSFC) under Project 6213000078, and in part by the NExT++ Research established by the National Research Foundation, Prime Minister's Office, Singapore, under its IRC@SG Funding Initiative. *(Corresponding author: Yong Cui.)*

Ziyi Wang, Yong Cui, and Xiaoyu Hu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: wangziyi0821@gmail.com; cuiyong@tsinghua.edu.cn; chaoese@gmail.com).

Xin Wang is with the Department of Electrical and Computer Engineering, State University of New York at Stony Brook, Stony Brook, NY 11794 USA (e-mail: x.wang@stonybrook.edu).

Wei Tsang Ooi is with the School of Computing, National University of Singapore, Singapore 117417 (e-mail: ooiwt@comp.nus.edu.sg).

Zhen Cao is with Huawei Technologies Company Ltd., Beijing 100085, China (e-mail: zhen.cao@huawei.com).

Yi Li is with Beijing Powerinfo Company Ltd., Beijing 100080, China (e-mail: tiger_li@263.net).

Digital Object Identifier 10.1109/TNET.2021.3129481

I. INTRODUCTION

LIVE streaming platforms, such as YouTube Live and Twitter's Periscope, have attracted millions of daily active users [1]–[3]. Since user engagement increases the revenue, these platform providers are increasingly interested in supporting interactive live streaming experience for their users, leading to *multi-party interactive live video streaming* as an emerging class of applications. In such an application, a user not only acts as a source of video, but also receives one or more streams from other users in the same session simultaneously. An example is collaborative talent show, where various geographically distributed online streamers can perform the arts of singing, acting, playing instruments, or other activities together and interact with each other by exchanging streams [4], [5]. Platforms that support such application (e.g., Inke.tv [6] and Douyu.tv [7]) have attracted hundreds of millions of users in recent years.

Three challenges arise from this new class of applications. First, applications such as collaborative talent show require a much tighter synchronization among the users. Schuett [8] reported a delay above 30 ms would disrupt the tempo, but a delay of up to 70 ms can be tolerated. This delay requirement is stricter than other multi-party live applications, such as multi-party video conferencing, where some existing work has achieved a tolerable delay of around 400 ms [9]. Such low tolerance to end-to-end delay means that the buffer occupancy (i.e., buffering delay) has to be kept small at the sender, server, and receiver. This, however, will increase the chance of buffer underflow and stall in the presence of network jitters and inaccurate throughput estimation. The challenge here is how to keep the buffer occupancy low without playback stall.

Second, given the heterogeneity of user devices and network conditions, it is important for a receiver to obtain a stream rate best suited for its requirement to maximize its quality of experience (QoE). There are many existing studies on receiver-driven adaptive bitrate (ABR) algorithms [10]–[15] that address this issue. Since each receiver is also a sender, however, a new question that arises here is: *at what bitrates should each sender encode its video stream to meet the requirement of the receivers?* Many existing solutions, in the context of multi-party video conferencing, rely on the use of a transcoding server (e.g., [16]–[19]), in which case the sender only needs to send a single stream at a high-enough bitrate and the server transcodes the stream to the required bitrate for the receiver. Such a solution not only requires additional

computation in the cloud with a higher infrastructure cost, but also introduces additional delay through the transcoding step. We therefore consider the scenario where the server only relays the stream without transcoding and the sender encodes the video into multiple streams at the bitrates required by the receivers. Since the uplink of a sender is likely a bottleneck, a sender can only generate a limited number of streams, which may not meet the needs of every receiver.

Third, for a receiver, the delay and quality requirements may differ for each sender. Users may have different QoEs or priorities, and set their preferences in advance. For instance, in a collaborative talent show, the end-to-end delay between a performer and a spectator can be higher than between two performers; This spectator may require higher video quality from the performers than that of other spectators. Each user has multi-dimensional requirements and interacts with other users. How to meet the requirements of all users at the same time becomes a challenge.

In this paper, we present a system for multi-party live streaming to address the challenges above. Our system has the following salient features. First, to minimize delay and stall duration, the streamer can increase or decrease the playback speed. Second, to more effectively utilize the uplink bandwidth, we adopt scalable video coding (SVC) to aggregate multi-rate streams and send them to the server, which then distributes different layers to different receivers. Third, we develop a QoE model that takes into account different requirements of multi-party interactive live streaming applications. By assigning weights to different terms, the system is able to personalize the preferences between each pair of users. Finally, we introduce the core component to this system, MultiLive, which is an adaptive bitrate control algorithm that is run centrally in the server, considering the various constraints in a many-to-many scenario as a non-linear programming problem that maximizes the overall QoE. MultiLive periodically solves the non-linear programming problem to obtain the target bitrate for each sender and each receiver and, in the interim, uses a feedback control algorithm to adjust the target bitrate to react to fluctuating throughput and variations in video encoding rate. The target bitrates are clustered into what the senders and the server actually send while minimizing the QoE loss. To the best of our knowledge, we are the first to design and implement an adaptive bitrate control algorithm for multi-party interactive live streaming that considers the many-to-many ABR problem while maximizing the QoE considering delay, smoothness, quality, and stall, holistically.

To make full use of the network bandwidth, we propose an available bandwidth measurement algorithm. It exploits the unique burst characteristic of keyframes during video streaming to passively probe the upper limit of the available bandwidth, based on which future bandwidth can be more accurately predicted to improve the effectiveness of ABR algorithm. This is different from existing studies that probe available bandwidth in a rapidly changing and unstable network environment [20], [21], where additional overhead is introduced to send dummy traffic or add a measurement proxy.

We implemented the ABR algorithm and the algorithm for the measurement of available bandwidth within an open-source video conferencing server, and evaluated their performance through extensive trace-driven simulations and test bed experiments. We collected and released, as an open dataset, more than 72 hours of uplink and downlink throughput measurements from live streaming servers.[1] In addition, Belgium 4G/LTE dataset [22] is used to test the performance. We also evaluated the algorithm on a live streaming platform with more than 2300 users. The results show that MultiLive outperforms the fixed bitrate algorithm, with 2-5$\times$ improvement in average QoE. When the loss rate is less than 0.5%, we achieved an end-to-end delay of 50 ms more than 90% of the time for the experiment on a fast network with propagation delay of 10 ms, meeting the delay requirement of 70 ms [8].

The rest of the paper is structured as follows. Section II presents the system architecture. Section III formulates the problem. Section IV elaborates the details of MultiLive. Section V introduces our technique for available bandwidth measurement. Implementation details are elaborated in Section VI. Simulation and real test bed evaluation results are presented in Section VII. Section VIII discusses the related work. Finally, we conclude the paper in Section IX.

II. System Architecture

We adopt SVC to provide flexibility in multi-party live streaming in the presence of heterogeneous user devices and bandwidths. Actually, there is a tradeoff to consider when choosing between server-based transcoding and sender-based SVC. First, transcoding tasks in the cloud need to use elastic computing resources, which increases the cost of the service providers as compared to the one with simple video relaying. The service provider has the motivation to assign the scalable encoding task to the streamer. As cloud transcoding constitutes a big portion of the cost of these providers, they have the incentives to offload and are actually using this approach in services. Second, with the popularity of live streaming, the streamers tend to become more and more professional with the use of many powerful devices and accessories [1]. Many streamers now use a special equipment to show the details of their video streaming sessions, and have a motivation to pay some more computing resources locally [23]. This is a new way the streamer is engaging with the system. In addition, some studies have found that current mainstream personal device processing speeds are already enough for real-time scalable encoding and decoding of multiple streams [24], [25].

We then present the architecture of our system (See Fig. 1). The three major logical entities are: the senders, the server, and the receivers. We only focus on the senders and receivers that participate in the same live streaming session here. In the current multi-party interactive live streaming applications, the number of streamers in a session is in the order of tens or less. Note that, despite our distinction of senders and receivers, in practice, the same streamer acts as both a sender and a receiver.

Each sender generates an SVC-coded, multi-rate, video stream and transmits it to the server. We adopt a *push*-based approach, where a frame is sent as soon as it is generated.

[1] https://github.com/STAR-Tsinghua/MultiLive_dataset

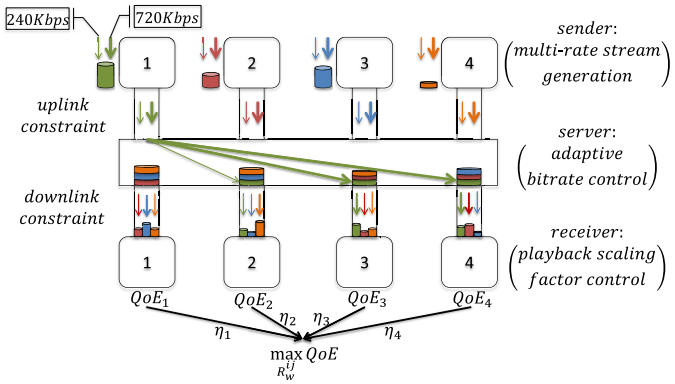


Fig. 1. The architecture of multi-party interactive live streaming.

This approach avoids the request-response overhead used in pull-based approach, commonly used in DASH-based video-on-demand streaming.

Upon receiving a frame, the server buffers it and relays the appropriate SVC layers of the frame to each receiver, also using the push-based approach. The decisions of: (i) what is the set of bitrates that each sender should produce, (ii) what is the bitrate that each receiver should receive, are determined by the server through an *adaptive bitrate controller*. The adaptive bitrate controller takes, as inputs, the uplink throughput of each sender, the downlink throughput of each receiver, the state of buffer occupancy at each receiver, and makes these decisions to maximize the total QoE of each receiver. The server also additionally collects the state information (buffer occupancy, downlink throughput, receiver's preferences for each sender) from each receiver and disseminates its decision (the set of bitrates) to the corresponding sender.

The receiver maintains a playback buffer for each sender. This partitioning of frames from each sender allows the receiver to manage the priorities and preferences across the senders (e.g., higher quality for a sender, lower delay for another). To avoid buffer under/overflow in live streaming, the receiver may adjust the playback speed. When the buffer occupancy rises above a high threshold, the receiver plays back the video at a faster rate to "catch up" and reduce the end-to-end delay of subsequent frames. When the buffer occupancy falls below a low threshold, the receiver plays back the video at a slower rate, to delay the onset of stalls and thus reducing the duration of stall. Previous studies found that playback rate changes up to 5% are imperceivable to most users and it is difficult for users to spot even much higher playback rate changes [26], [27]. Thus, without degrading the users' quality of experience, our dynamic playback adjustment can reduce the latency as well as avoid rebuffering.

As noted above, we adopt a frame-level granularity in transmission. This approach is commonly used in DASH-based live streaming system using chunk-encoding in Common Media Application Format (CMAF). Transmitting, buffering, and playing back at a frame-level granularity, instead of segment-level granularity commonly used in video-on-demand systems, keeps the end-to-end delay small.

To summarize our design decisions, our system architecture supports the following objectives: First, by processing at a frame-level granularity, adopting push-based transmission, increasing playback speed when needed, and avoiding transcoding at the server, the delay is kept small. Second, using SVC, the sender sends only a single stream to meet the bandwidth requirements of multiple heterogeneous receivers. Finally, by optimizing the QoE (through finding the best bitrate configuration), we maintain a high quality of experience for the users. This final component is the focus for the rest of this paper. In the next section, we will first present some preliminaries and state the optimization problem. This presentation is followed by Section IV, where we will present how the system optimizes the QoE.

III. Problem Formulation

In this section, we present a model of the network constraint, buffer occupancy, and QoE. We end this section with a statement of the optimization problem to be solved at the server to maximize the QoE. To facilitate our presentation, the major notations are summarized in Table I.

A. Network Constraint

Let $C_w^{up(i)}$ denote the average uplink throughput when sender i sends the w-*th* frame and $C_w^{down(j)}$ denote the average downlink throughput when receiver j receives the w-*th* frame. Let R_w^{ij} denote the bitrate of the w-*th* frame from sender i to receiver j.

Considering the uplink of the sender i, the total throughput of the streams it generates must be less than the limit of the uplink throughput after aggregating the streams with SVC. Considering the characteristics of clustering and SVC, the constraints of the uplink can be relaxed as:

$$\max_j \left\{R_w^{ij}\right\} \leq C_w^{up(i)}. \tag{1}$$

Similarly, considering the downlink of the receiver j, the sum of the bitrates of several streams it receives must be less than the total downlink throughput:

$$\sum_i R_w^{ij} \leq C_w^{down(j)}. \tag{2}$$

B. Buffer Model

The buffer in this paper refers to the queue of video frames to be consumed, used mainly to cope with network jitters. The senders, server, and receivers each have one or more buffers. Among them, the receiver's buffers are directly related to the playback condition, and are more important for the QoE of the receiver. So we focus on the receiver's buffers in our model.

The receiver's playback buffer contains video frames that have been received but yet to be played back. We let $B_w^{down(ij)}$ be the receiver buffer occupancy (in unit of time) when receiver j starts to receive the w-*th* frame of sender i. Let $C_w^{down(ij)}$ be the average downlink throughput when receiver j receives the w-*th* frame of i. Let S_w^{ij} be the size of the w-*th* frame and D_w^{ij} be the duration of the w-*th* frame. The time taken to fully receive the w-*th* frame is $S_w^{ij}/C_w^{down(ij)}$. The buffer occupancy increases by D_w^{ij} seconds after the w-*th*

TABLE I
MAJOR NOTATIONS USED IN THIS PAPER

Notation	Description
$C_w^{up(i)}$	Average uplink throughput when *i* sends the *w-th* frame
$C_w^{down(j)}$	Average downlink throughput when *j* receives the *w-th* frame
$C_w^{down(ij)}$	Average downlink throughput when *j* receives the *w-th* frame of *i*
$B_w^{down(ij)}$	Current receiver buffer occupancy when *j* starts to receive the *w-th* frame of *i*
$\hat{B}_w^{down(ij)}$	Target receiver buffer occupancy when *j* starts to receive the *w-th* frame of *i*
$B_{fast}^{down(ij)}$	Buffer occupancy threshold of fast playback from *i* to *j*
$B_{slow}^{down(ij)}$	Buffer occupancy threshold of slow playback from *i* to *j*
$T_w^{up(i)}$	System time when *i* starts to generate the *w-th* frame
$T_w^{down(ij)}$	System time when *j* starts to receive the *w-th* frame of *i*
R_w^{ij}	The real bitrate of the *w-th* frame from *i* to *j*
$\hat{R}_w^{ij}$	The target bitrate of the *w-th* frame from *i* to *j*
S_w^{ij}	The size of the *w-th* frame from *i* to *j*
D_w^{ij}	The duration of the *w-th* frame from *i* to *j*
P^{ij}	The playback scaling factor from *i* to *j*
Q_K^{ij}	Cumulative video quality of *K* frames from *i* to *j*
V_K^{ij}	Cumulative video quality variations of *K* frames from *i* to *j*
E_K^{ij}	Cumulative rebuffer duration of *K* frames from *i* to *j*
L_K^{ij}	Delay of *K* frames from *i* to *j*

frame is received and decreases as the receiver plays back the video. So the evolution of the receiver buffer occupancy level can be derived as:

$$\hat{B}_{w+1}^{down(ij)} = \left(B_w^{down(ij)} - \frac{S_w^{ij}}{C_w^{down(ij)}} \right)_+ + D_w^{ij}, \quad (3)$$

where $\hat{B}_{w+1}^{down(ij)}$ is the target buffer occupancy when receiver j starts to receive the next frame and $(x)_+ = max\{x, 0\}$. Note that if $B_w^{down(ij)} < S_w^{ij}/C_w^{down(ij)}$, the buffer will become empty while the receiver is still receiving the w-*th* frame, leading to stalls (i.e., the receiver's playback buffer does not have frames to render). So the first term in the equation above cannot be negative.

In addition, we consider adjusting the playback speed to achieve fine-grained delay control, according to the current buffer occupancy $B_w^{down(ij)}$ and two thresholds: $B_{fast}^{down(ij)}$ and $B_{slow}^{down(ij)}$. When the buffer occupancy is more than the fast playback threshold, $B_w^{down(ij)} > B_{fast}^{down(ij)}$, the receiver plays back faster to reduce the delay; When the buffer occupancy is less than the slow playback threshold, $B_w^{down(ij)} < B_{slow}^{down(ij)}$, the receiver slows down the playback to alleviate stall. Let P^{ij} be the scaling factor that controls the video playing speed from the sender i to the receiver j, with $P^{ij} = 1$ means normal playback. Then the actual duration of video to playback per second is P^{ij}. The duration of video received from the downlink per second is $C_w^{down(ij)}/R_w^{ij}$. The net consumption rate of video duration is the difference between the two values. After fast/slow playback, the video delay changes from L_w^{ij} to $L_{w'}^{ij}$. The catch-up time we need is $(L_w^{ij} - L_{w'}^{ij})/(P^{ij} - 1)$. Multiplying the catch-up time by the net consumption rate is the amount of change in the buffer. So the evolution of the receiver buffer occupancy level considering playback speed adjustment ($P^{ij} \neq 1$) can be derived as:

$$\hat{B}_{w'}^{down(ij)} = B_w^{down(ij)} - \left(P^{ij} - \frac{C_w^{down(ij)}}{R_w^{ij}} \right) \cdot \frac{L_w^{ij} - L_{w'}^{ij}}{P^{ij} - 1}. \quad (4)$$

Eq. (4) is used to predict the buffer occupancy only if the fast/slow playback occurs. The playback scaling factor P^{ij} can be flexibly selected according to actual conditions. In the case of normal playback, Eq. (3) is used.

C. QoE Model

In multi-party interactive live streaming, the overall session QoE should be considered at two different levels: (i) the session QoE should be the weighted sum of each receiver's QoE, and (ii) a receiver's QoE then depends on the factors in which it receives data from others. We mainly refer to the QoE defined by Yin *et al.* [14] and Ahmed *et al.* [28]. Specifically, each receiver's QoE includes the following four aspects:

(1) Cumulative video quality Q_K^{ij}: Let $q(\cdot)$ be a non-decreasing function that maps bitrate R_w^{ij} to the perceived video quality $q(R_w^{ij})$. Then the cumulative video quality of K consecutive frames from sender i to receiver j is:

$$Q_K^{ij} = \sum_{w=1}^{K} q(R_w^{ij}). \quad (5)$$

(2) Cumulative video quality variations V_K^{ij}: From the receiver's perspective, frequent bitrate switching is undesirable. Therefore, the QoE model should add video quality variations as penalty. The cumulative video quality variations of K consecutive frames sent from sender i to receiver j is:

$$V_K^{ij} = \sum_{w=1}^{K-1} |q(R_{w+1}^{ij}) - q(R_w^{ij})|. \quad (6)$$

(3) Cumulative stall duration E_K^{ij}: When the buffer is drained out, a stall occurs and deteriorates the receiver's QoE. Therefore, the stall duration should also be a penalty in the QoE model. The cumulative stall duration of K consecutive frames sent from sender i to receiver j is:

$$E_K^{ij} = \sum_{w=1}^{K} \left(\frac{S_w^{ij}}{C_w^{down(ij)}} - B_w^{down(ij)} \right)_+. \quad (7)$$

(4) Delay L_K^{ij}: Video-on-demand (VoD) streaming has a more relaxed requirement of delay and can use a large

playback buffer, whereas live streaming cannot. To maintain interactivity, the most important requirement is low delay [29]. Therefore, the QoE model should also add delay as penalty. Let $T_w^{up(i)}$ denote the system time when sender i starts to generate the w-*th* frame and $T_w^{down(ij)}$ denote the system time when receiver j starts to receive the w-*th* frame of sender i. Then the delay of after sending K consecutive frames from sender i to receiver j is:

$$L_K^{ij} = T_K^{down(ij)} - T_K^{up(i)} + B_K^{down(ij)}. \tag{8}$$

Since receivers have different preferences for the above four aspects, we define the receiver j's QoE as the weighted sum of the above four aspects, namely:

$$QoE_j = \sum_i \left(\alpha_{ij} Q_K^{ij} - \beta_{ij} V_K^{ij} - \gamma_{ij} E_K^{ij} - \delta_{ij} L_K^{ij}\right), \tag{9}$$

where $\alpha_{ij}, \beta_{ij}, \gamma_{ij}$, and δ_{ij} are the weights of the different QoE terms between sender i and receiver j. Note that these weights are per sender-receiver pair, allowing each receiver to personalize its QoE preference to different senders depending on the amount of interaction needed and the context of the application. Finally, the overall session QoE is the weighted sum of all receivers' QoE:

$$QoE = \sum_j \eta_j QoE_j, \tag{10}$$

where η_j is the weight of the j-*th* receiver.

The server obtains global state information and calculates the bitrates that each sender should produce. Then it informs senders of these information. Senders generate the streams as required. So the problem is, to maximize the global QoE, how to design this adaptive bitrate control algorithm running on the server? That is, given the current buffer occupancy and uplink/downlink throughput prediction, how many streams are generated by each sender and what are their real bitrates R_w^{ij}? The problem can be formulated as:

$$\begin{aligned} &\max_{R_w^{ij}} \ QoE \\ &s.t. \ (1)\ (2)\ (3)\ (4) \end{aligned} \tag{11}$$

IV. Algorithm Design

In this section, we first give an overview of the algorithm, namely MultiLive. Then we elaborate the solution in three subsections, including non-linear programming, buffer feedback adjustment, and bitrate clustering.

A. Design Overview

The MultiLive algorithm workflow is shown in Fig. 2. To find the number of streams and the bitrate of each stream R_w^{ij} a sender i can transmit to the receiver j, we split the solution into two steps. First, we calculate the target bitrate $\hat{R}_w^{ij}$ for each pair of sender-receiver. Both the receivers and senders have to jointly decide which bitrate to produce and which bitrate to receive. Specifically, we build a non-linear programming (NLP) solution to get the target bitrate $\hat{R}_w^{ij}$. It is also updated through the buffer feedback adjustment (BFA) to alleviate system errors. NLP and BFA run in coarse- and fine-grained manner respectively to meet both the high accuracy and the real-time constraint. As we will introduce in the evaluation section, we run NLP with a period of 2000 ms and BFA with a period of 200 ms in our system. Second, we cluster and aggregate the target bitrate $\hat{R}_w^{ij}$ to the real bitrate R_w^{ij} to transmit according to the SVC requirements.

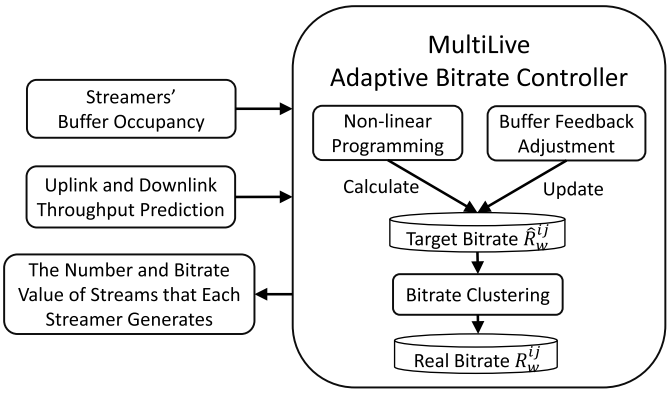


Fig. 2. The algorithm workflow of adaptive bitrate controller.

B. Non-Linear Programming (NLP)

In a multi-party scenario, finding the target bitrate is an optimization problem that takes into account various constraints from a global perspective, as we have formulated in Eq. (11). Previous studies [30], [31] show that as the bitrate increases, the rate of increase in video quality score decreases. In other words, there is no linear correlation between the bitrate of a video stream and its perceptual quality. Guo *et al.* [18] used logarithmic function to characterize the relationship between video quality and video bitrate. Based on these studies, the video quality in our QoE objective function is set to be logarithmic. Also, the target bitrate is calculated on a continuous domain rather than discrete.

For a constrained non-linear programming problem, the constraints can be converted to penalty to turn the problem into an unconstrained one to solve by the gradient descent method. We build a non-linear programming (NLP) solution to solve Eq. (11) and get preliminary result (target bitrate $\hat{R}_w^{ij}$). Denoting N as the number of streamers, the time complexity of NLP solution is $O(N^{5.614})$. The reason is that matrix inversion will not be more complicated than matrix multiplication [32] and the complexity of the commonly used multiplication algorithm (i.e., Strassen's algorithm) is $O(M^{2.807})$ for matrices of M dimension [33]. Although the problem can be solved in polynomial time complexity, it is not fast enough for a real-time update. So we need a complementary approach to calculate the target bitrate faster. Furthermore, the input parameters are not always accurate due to factors, such as inaccurate throughput estimates and fluctuations in video encoding rate. Therefore, we introduce an additional step, buffer feedback adjustment, which updates the target bitrate based on the buffer state to alleviate the input errors. We will introduce it in the next section.

C. Buffer Feedback Adjustment (BFA)

Since the change of buffer occupancy reflects the change of throughput, we can make some feedback adjustments to the target bitrate $\hat{R}_w^{ij}$. In this way, we may reduce the throughput prediction errors and make target bitrate more accurate. In Eq. (3), given the buffer and throughput when the receiver starts to receive the w-*th* frame, we can estimate that the buffer occupancy of the $(w+1)$-*th* frame when the throughput is unchanged. However, at the moment when the receiver actually starts to receive the $(w+1)$-*th* frame, we can get the actual buffer occupancy. The difference between the target value and the real value reflects the change rate of the throughput. It determines the range of target bitrate adjustment.

To perform the adjustment, we refer to the PID controller [34], a widely used feedback control technique. It includes proportional controller, integral controller and derivative controller. It monitors the error value e_t, which is the difference between the target value and the real value. Then it can output the control signal u_t:

$$u_t = K_p e_t + K_i \int_0^t e_\tau d\tau + K_d \frac{de_t}{dt}, \tag{12}$$

where the three parameters K_p, K_i, and K_d represent the coefficients for the proportional, integral, and derivative terms respectively. The derivative term is sensitive to the measurement noise [35]. So we make some modifications to suit our specific scenario. In our control policy, the parameter for the derivative control K_d equals zero. So strictly speaking, our controller is a PI controller. The remaining two terms are as follows:

(1) Proportional controller calculates the difference between the real buffer and the target buffer to alleviate the prediction errors. We use Z_p to represent this term:

$$Z_p = K_p \left(B_{w+1}^{down(ij)} - \hat{B}_{w+1}^{down(ij)} \right). \tag{13}$$

(2) Integral controller integrates the difference between the real buffer and the target buffer to alleviate cumulative system errors. We use Z_i to represent this term:

$$Z_i = K_i \int_0^{w+1} \left(B_t^{down(ij)} - \hat{B}_t^{down(ij)} \right) dt. \tag{14}$$

Therefore, we obtain the updated target bitrate value based on the following buffer feedback adjustment (BFA):

$$\hat{R}_{w+1}^{ij} = \hat{R}_w^{ij} + Z_p + Z_i. \tag{15}$$

In our case, the control signal u_t represents the target bitrate. The parameters K_p and K_i are responsible for completing the dimension conversion. We need to update the target bitrate for each pair of streamers using PI controller. So the time complexity of the buffer feedback adjustment algorithm is $O(N^2)$.

D. Bitrate Clustering

Through the non-linear programming solution and the buffer feedback adjustment, we can get the target bitrate $\hat{R}_w^{ij}$ for each pair of sender-receiver. The sender, however, may not have the encoding capacity or uplink bandwidth to encode and transmit at each of these target bitrates. To alleviate the problem, the server clusters the target bitrates $\hat{R}_w^{ij}$ to obtain the actual bitrates R_w^{ij}, according to which the sender produces the sub-streams and uses the SVC to aggregate them into one stream. This approach reduces the overhead of encoding and transmission.

Algorithm 1 Bitrate Clustering

Input: N: the number of senders;
$\hat{R}_w^{ij}$: the target bitrate of the w-*th* frame from i to j;
m_i: the number of streams generated by sender i
Output: $\mu_1^i, \mu_2^i \cdots \mu_{m_i}^i$: the cluster centroids of sender i

```
Initialize cluster centroids μ_1^i, μ_2^i ··· μ_{m_i}^i randomly
for i = 1 to N do
  repeat
    for j = 1 to N and j ≠ i do
      class(R̂_w^ij) ← arg min_k |QoE(R̂_w^ij) − QoE(μ_k^i)|
    end for
    for k = 1 to m_i do
      μ_k^i ← (Σ_j 1(class(R̂_w^ij)=k) R̂_w^ij) / (Σ_j 1(class(R̂_w^ij)=k))
    end for
  until convergence
end for
return μ_1^i, μ_2^i ··· μ_{m_i}^i
```

In the process of clustering, we define *QoE loss* as the difference between the QoE value of the target bitrate and the QoE value of the cluster centroid bitrate. The ideal situation is that each sender produces an SVC stream that minimizes overall QoE loss of receivers. We use the *K*-means clustering algorithm for clustering. The details are shown in Algorithm 1. Ideally, if the cluster centroids in the several consecutive iterations are the same, the algorithm is said to have converged. But in practice, we use a less strict criteria for convergence: Given a threshold σ, for the cluster centroid μ_k^i in an iterative process, if the $(\mu_k^i)'$ produced by the next iteration satisfies $\sigma < \frac{(\mu_k^i)'}{\mu_k^i} < \frac{1}{\sigma}$ and this state can be maintained for several rounds, we consider the algorithm to have converged. The returned $\mu_1^i, \mu_2^i \cdots \mu_{m_i}^i$ (the cluster centroids of sender i) are the actual bitrates that the sender i needs to encode the video into. The server notifies senders of the information. Then the senders generate the streams as required. Considering that the number of streamers in a session is in the order of tens or less, this algorithm typically needs dozens of iterations to converge. The time complexity of the bitrate clustering using *K*-means algorithm [36] is $O(N^2)$.

MultiLive periodically solves the non-linear programming problem to obtain the target bitrate and, in the interim, uses a feedback control algorithm to adjust the target bitrate. Then the target bitrates are clustered into real bitrates. So the time complexity of the overall algorithm is $O(N^{5.614} + N^2) = O(N^{5.614})$ (N is the number of streamers). In our scenario, N is in the order of tens or less. So this time complexity is acceptable.

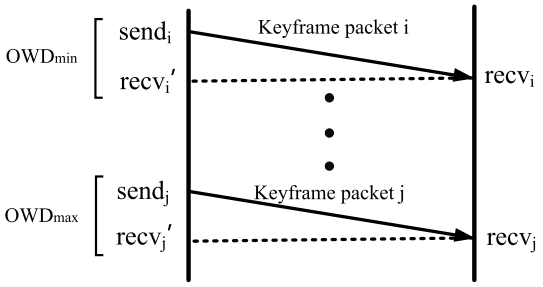


Fig. 3. The available bandwidth measurement algorithm.

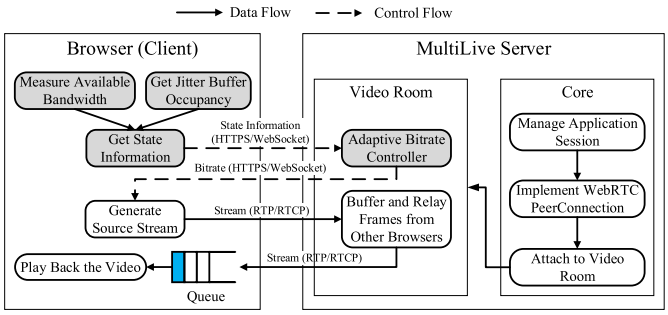


Fig. 4. Implementation architecture (The gray part in the figure is the module we added).

V. Available Bandwidth Measurement

An important input of the adaptive bitrate control algorithm is the prediction of future bandwidth. Previous studies mostly focus on optimizing the prediction process, using time-series model [12], [14], [37], machine-learning model [38], [39], and data-driven model [40]. However, as the historical throughput is generally lower than the available bandwidth, it will result in a lower predicted bandwidth value. On the other hand, literature studies either send dummy traffic [20] or add a measurement proxy [21] to probe available bandwidth in rapidly changing and unstable mobile Internet, which increase the overhead and difficulty of deployment in real-world scenarios. Furthermore, throughput measurement is accurate only in cases where the pipeline is filled. Although a segment downloading in the pre-recorded video streaming could fill the pipeline, in live streaming scenarios, the sender sends the packets only after a video frame is generated. The predicted value would be lower if the available bandwidth is estimated based on the throughput measured.

We exploit the burst characteristic of keyframe in the video streaming to passively probe the upper limit of the available bandwidth without causing any transmission overhead. Specifically, in multi-party live streaming, each coded video stream consists of some keyframes and many non-keyframes. Keyframe, also known as the intra-frame (I-frame), is a frame that is independently compressed without referencing other frames. It is the least compressible but does not depend on other video frames to be decoded. Compared to a keyframe, non-keyframe depends on data from previous frames for decoding and is more compressible. Generally, a keyframe is larger in size and sent as a burst. We thus choose to use keyframe burst to probe the bandwidth passively.

The idea of the algorithm is illustrated in Fig. 3. A keyframe is transmitted as a burst with dozens of packets. We assume that all packets in the same keyframe have the same length (i.e. Maximum Transmission Unit). The algorithm first collects the One-Way Delay (OWD) of all packets belonging to a keyframe. An one-way delay is mainly composed of 4 parts: processing delay (time to examine the packet), queuing delay (time to wait at output link for transmission), transmission delay (time to push the packet's bits into the link), and propagation delay (time to propagate on the physical link) [41]. All packets of the same keyframe generally experience the same processing delay, transmission delay and propagation delay. The only difference is the queuing delay. The algorithm finds the packet i with the minimum one-way delay (i.e., the packet at the head of the burst) and the packet j with the maximum one-way delay (i.e., the packet at the end of the burst). The difference between OWD_{max} and OWD_{min} is the difference between the queuing delay of packet i and packet j, which is also the accumulated transmission time of the entire keyframe from the network port to the link. Dividing the total size in bytes of the keyframe by this difference reflects the available bandwidth. Specifically, let the sending timestamp of packet i be $send_i$. The packet arrival timestamp at the receiver end is $recv_i$, and the corresponding time of the sender is $recv'_i$. The sending delay of the keyframe is thus:

$$\begin{aligned}\Delta OWD &= OWD_{max} - OWD_{min} \\ &= (recv'_j - send_j) - (recv'_i - send_i) \\ &= [(recv_j - send_j) + (recv'_j - recv_j)] \\ &\quad -[(recv_i - send_i) + (recv'_i - recv_i)] \\ &= (recv_j - send_j) - (recv_i - send_i). \end{aligned} \quad (16)$$

Since both the values of OWD_{max} and OWD_{min} contain the clock difference between the sender and the receiver, the use of subtraction can easily eliminate the difference, thus avoiding the clock synchronization problem. In addition, this also converts the inputs of ΔOWD calculation into easily available data to estimate the available bandwidth:

$$\begin{aligned} Avai\_bwd &= \frac{Data_{keyframe}}{\Delta OWD} \\ &= \frac{Data_{keyframe}}{(recv_j - send_j) - (recv_i - send_i)}. \end{aligned} \quad (17)$$

With the use of the burst data of keyframe to probe the available bandwidth, we can improve the effectiveness of ABR algorithm so that the network bandwidth can be more efficiently utilized.

VI. Implementation

We implement the ABR algorithm and the available bandwidth measurement algorithm within an open-source video conferencing server, which is based on WebRTC [42], [43]. The implementation architecture is shown in Fig. 4. To enable the server to obtain the state information of each browser, we modify the browser code, the server code, and the communication code between the browser and the server. Specifically, we add the available bandwidth measurement module and jitter

buffer occupancy acquisition module in the browser. We also add the adaptive bitrate controller module in the server. After getting the state information, the browser sends it to MultiLive server via HTTPS/WebSocket. The server collects the state information from multiple browsers, runs the ABR algorithm through the adaptive bitrate controller, decides the bitrate that the browser should generate, and returns this result to each browser. Then the browsers generate streams as required.

In terms of media data, the browser encodes the stream through WebRTC VP9-SVC encoder and uses the RTP/RTCP protocol to transmit packets to the server. The server buffers and relays media data packets from other browsers to this browser. After decoding the stream through SVC decoder, the browser plays back the video. In addition, a core module is used in the server to be responsible for three things: (i) managing application sessions with browsers through a REST-ful API; (ii) implementing the WebRTC communication with the same browsers, by taking care of the whole WebRTC life cycle (negotiation, establishment and management of connections between users); (iii) attaching to video room, in order to allow them to exchange messages and more importantly media data.

As for compatibility, enterprise networks with HTTP proxy/firewall create substantial problems with WebRTC, which is running over RTP with UDP, since it was designed to do so. This problem has been recognized by both WebRTC and IETF community, with many standard and proprietary solutions [44]–[46]. First of all, when RTP/UDP fails to establish a connection, some clients will run RTP over TCP, and this is permitted in some cases and standardized by RFC 4571 [45]. This solution has been implemented in many commercial products as we asked around. However, this is not perfect because there is still a problem when the proxy checks the layer-7 HTTP information. So, the Janus WebRTC proposes to use HTTP to tunnel the RTP payload [46], which is a neat and straightforward solution. In this way, RTP flows can be tunneled over HTTP, which will make our system compatible with the Internet middleboxes.

VII. Evaluation

We conducted extensive trace-driven simulations and real test bed experiments to evaluate our method. To obtain the throughput traces from an actual deployed live streaming service, we collected the uplink and downlink data from three geographically distributed live streaming servers for more than 72 hours. We refer to this as the *Commercial Dataset*. We also use the Belgium 4G/LTE dataset [22] in the evaluation. This dataset consists of throughput measurements in 4G networks along several paths in and around the city of Ghent, Belgium. We distributed the two types of traces to five streamers in the simulator respectively and generated a frame sequence at a rate of 30 frames per second. We also performed a test bed evaluation and deployed the algorithm on a real live streaming platform.

The parameters we used in our experiments are as follows. We set the buffer thresholds for faster and slower playback, $B_{fast}^{down(ij)}$ and $B_{slow}^{down(ij)}$ to 90 ms and 30 ms uniformly for each sender i and each receiver j. In addition, we set the number of clusters used in K-means algorithms to 2, considering there are only five streamers in our settings. For the QoE model, we set $\alpha_{ij} = 1, \beta_{ij} = 1, \gamma_{ij} = 1$, and $\delta_{ij} = 20$ (strict requirement for the delay term), for all i and j, except for $\alpha_{0,2} = 0.6$ and $\delta_{0,2} = 28$ (prefers lower quality but lower delay); $\alpha_{1,3} = 1.2$ and $\delta_{1,3} = 16$ (prefers higher quality but higher delay) to illustrate the different preferences for receivers 2 and 3.

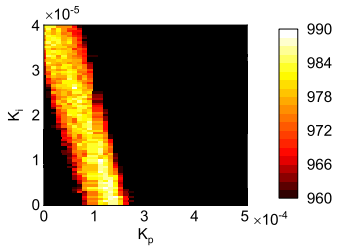


Fig. 5. Heat map for the proportional controller coefficient K_p and integral controller coefficient K_i.

A. Parameter Choice of PI Controller

In our first experiment, we study the sensitivity of the method to the parameters K_p and K_i of the PI controller. Since the PI controller is used to adjust the buffer feedback thus the target bitrate, if parameters K_p and K_i are sensitive to the network environment, tuning them will require more efforts. So we want to explore whether there exist a set of K_p and K_i values that work well in a wide range of network conditions. We consider the network throughput from the 72 hours traces separately. For the *k-th* hour network trace, we vary the values of K_p and K_i in a large range to obtain the corresponding QoE values. Then we consider all the network traces and accumulate the QoE values. Fig. 5 shows the heat map with the heat for each pair of K_p and K_i values as the average QoE value. A larger heat value means that it leads to good performance for more traces of network throughput. For different K_p and K_i pairs, the heat value varies. The white region represents the highest values, indicating that the corresponding K_p and K_i pairs provide good performance across almost all network traces. The recommended ranges for the proportional controller coefficient K_p and integral controller coefficient K_i are:

$$\begin{aligned} K_p &\in [1.2 \times 10^{-4},\ 2.0 \times 10^{-4}] \\ K_i &\in [0.2 \times 10^{-5},\ 1.0 \times 10^{-5}] \end{aligned} \tag{18}$$

Considering that the network traces are collected from a real live streaming server and that they exhibit different temporal and spatial characteristics, the results show that K_p and K_i can be tuned to accommodate the large variations among different traces. It means that we can find a range of K_p and K_i values to make the PI controller practical. We use $K_p = 1.2 \times 10^{-4}$ and $K_i = 1.0 \times 10^{-5}$ as our default settings.

B. Interval Choice of NLP and BFA

The two major steps in calculating the target bitrate are non-linear programming (*NLP*) and buffer feedback

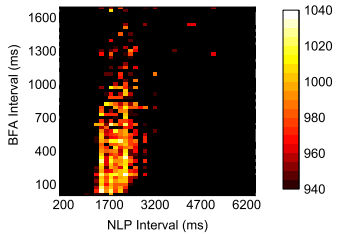


Fig. 6. Heat map for non-linear programming (*NLP*) and buffer feedback adjustment (*BFA*) interval.

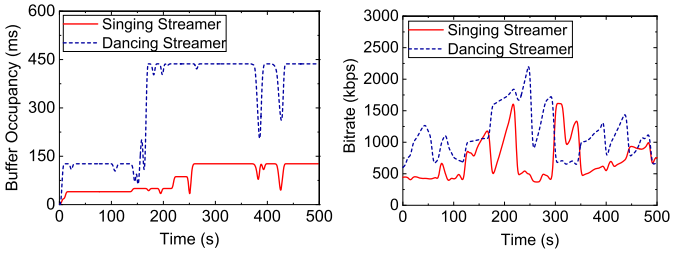


Fig. 7. Receiver buffer occupancy and bitrate from singing and dancing streamers, respectively.

adjustment (*BFA*). The importance of *NLP* is to provide the consideration of global constraints and resource competition among streamers. It relies on the throughput estimation. In the meantime, *BFA* is used to alleviate system errors during the modeling and measurement process. They can be executed at different intervals (I_{NLP}, I_{BFA}), resulting in different QoE effects. We want to explore whether there exist a set of I_{NLP} and I_{BFA} values that work well in a wide range of network environment. We also consider 72 hours of network throughput traces separately. For the k-*th* hour network trace, we vary the values of I_{NLP} and I_{BFA} in a large range to obtain the corresponding QoE values. Then we consider all the network traces and get the average QoE values. Fig. 6 shows the heat map. A larger heat value means that it leads to good performance for more traces of network throughput. The recommended ranges for *NLP* and *BFA* interval are:

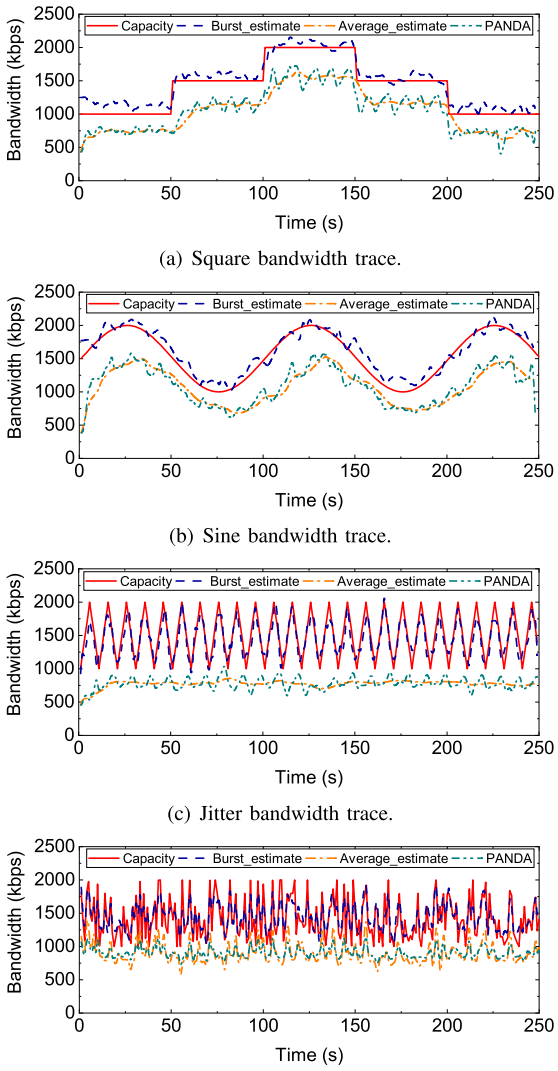
$$I_{NLP} \in [1400\,ms,\ 2300\,ms]$$
$$I_{BFA} \in [50\,ms,\ 400\,ms] \tag{19}$$

If the time interval of *NLP* is short, *NLP* will frequently generate target bitrates with measurement errors, leaving *BFA* little time to adjust the bitrate based on feedback, resulting in a low QoE value. On the other hand, if the time interval of *NLP* is long, the server cannot response to changes quick enough. While in live streaming, *BFA* is a relatively conservative strategy. It is hard to increase the bitrate once it is lack of global information, which leads to a low QoE value too. In the mean time, QoE value falls as the interval of *BFA* grows because a long interval of *BFA* leads to the insensitivity to buffer changes. We use $I_{NLP} = 2000$ ms and $I_{BFA} = 200$ ms as our default settings.

C. Preferences of Multiple Streamers

To illustrate how effective we can adjust to different preferences of steamers, we consider the scenario with three streamers i, j, and k. Streamers i and j are singing together in a chorus; Streamer k is dancing for them. Streamer i hopes the stream delay of j who sings with him is low. So i can increase the delay penalty weight in the QoE model. Streamer i also wants to see the dancing posture of k clearly. So i can increase the video quality weight of k in the QoE model. Dynamically setting the weights of different aspects in the QoE model for different streamers can effectively satisfy user preferences.

We simulate the above scenario in the simulator. Streamer i increases the delay penalty weight of j and the video quality weight of k. We measure i's receiver buffer occupancy, which stores frames sent from j and k respectively. We also measure bitrates from these two streamers. The result, shown in Fig. 7, shows that after the weight setting, the receiver buffer occupancy levels from two streamers have a great gap. The buffer storing j's data is obviously lower to maintain a lower delay while the buffer storing k's data is obviously higher to maintain a higher bitrate. Setting different weights for QoE model changes the receiver buffer occupancy. On the other hand, bitrates from two streamers also show differences.



(a) Square bandwidth trace.

(b) Sine bandwidth trace.

(c) Jitter bandwidth trace.

(d) Real bandwidth trace.

Fig. 8. The performance of available bandwidth measurement algorithm under different bandwidth traces.

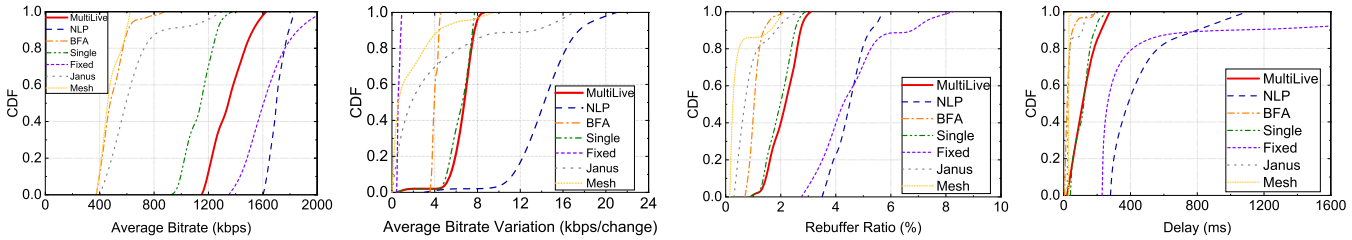


Fig. 9. Detailed performance using the commercial dataset.

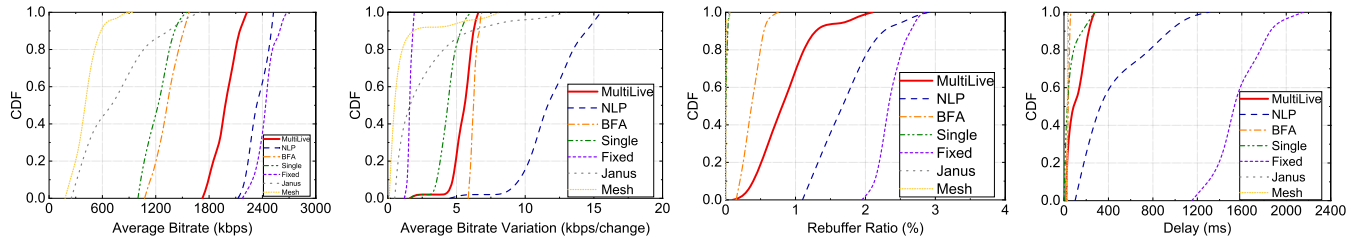


Fig. 10. Detailed performance using the Belgium 4G/LTE Dataset.

The bitrate from k is relatively high to ensure high video quality, and the bitrate from j is lower to ensure the smoothness and low delay. In fact, each streamer can set different priority for each stream he receives, including high bitrate, low bitrate switching, low stall duration, low delay and balanced. Personalized requirements can be well satisfied.

D. Available Bandwidth Measurement

To evaluate the performance of the available bandwidth measurement algorithm, we perform experiments on the test bed. The client and the server modules are implemented on two Linux machines respectively. The client connects to the server via 802.11ac WiFi. We use `tc` (traffic control, a user-space utility program used to configure the Linux kernel packet scheduler) [47] to tune the network condition to simulate four different variations of network bandwidth, including square, sine, jitter and real bandwidth traces. In these four different network situations, we insert timestamps into the code to get one-way delay, and analyze the RTP packets with the same timestamp to get the keyframe size. Then the client estimates the bandwidth according to the formula $Data_{keyframe}/\Delta OWD$ which we introduce above.

The results are shown in Fig. 8. The $Burst\_estimate$ line represents the bandwidth estimated with our algorithm, which applies the data amount in keyframe to divide by the difference between the maximum OWD and the minimum OWD. The $Average\_estimate$ line obtains the estimated bandwidth using common algorithm to find the amount of data received per second, and then performs harmonic average smoothing. The $PANDA$ line adopts additive-increase-multiplicative-decrease (AIMD) principle to probe available bandwidth based on the network throughput [48]. In all four different network scenarios, the $Burst\_estimate$ line fits well with the $Capacity$ line (bandwidth capacity limited by `tc`). In $Average\_estimate$, low rate non-keyframes are also used in the estimation, resulting in a smaller value. The $PANDA$ line adopts AIMD based on the $Average\_estimate$ line. So it is roughly similar to the $Average\_estimate$ line with some fluctuations. In addition, when the network fluctuates, $Burst\_estimate$ can always well follow the changes of the network in time, while $Average\_estimate$ and $PANDA$ have a certain delay and have a poor ability to follow the changes of the network. For a real bandwidth trace from a *Commercial Dataset* which contains sudden bursts and drops, $Burst\_estimate$ can measure the bandwidth well with the use of the burst data of keyframe. If we define $1 - \frac{|C-\hat{C}|}{C}$ as the accuracy (C and $\hat{C}$ are the actual and estimated available bandwidth respectively), then the average accuracies of our algorithm in the square, sine, jitter and real cases are 92.6%, 93.2%, 89.8% and 91.9% respectively. The accuracies of $Average\_estimate$ algorithm are 74.4%, 70.0%, 53.8% and 64.5%. In addition, the accuracies of $PANDA$ algorithm are 74.3%, 70.1%, 54.1% and 65.8% in these four cases. This shows that our probing algorithm can much more accurately estimate the available bandwidth and well track the changes under all test scenarios.

E. QoE Performance

We now evaluate the performance of MultiLive in terms of QoE and its four components, using the following methods as baselines:

- *NLP*: A simpler version of *MultiLive* where *BFA* is not performed.
- *BFA*: Another simpler version of *MultiLive* where *NLP* is not performed.
- *Single*: A simpler version of *MultiLive* where only a single bitrate is generated by each sender.
- *Fixed*: Computes the bitrates using initial network conditions, then continues to send at these bitrates without adapting to changing network conditions.
- *Janus*: A bitrate control algorithm in an open-source video conferencing server based on WebRTC [42], [43].

It comprehensively considers sender-side bandwidth estimation and receiver-side RTCP REMB feedback message.

- *Mesh*: Each streamer establishes a point-to-point (p2p) connection with all other streamers directly. It divides the uplink bandwidth equally and generates several same bitrate streams for multiple receivers [49].

Fig. 9 and Fig. 10 present the performance of average bitrate, average bitrate variation, stall ratio (stall time/total time) and delay in the form of CDF for each dataset.

(i) *NLP*: With the estimation of global throughput and consideration of QoE weights, *NLP* takes more advantage of the network transmission capacity to achieve high bitrate. However, the adjacent two bitrate decisions use separate throughput data, resulting in large bitrate jitter. In addition, it is called at a larger interval because of its overhead on communication and computation. So it cannot provide a fine-grained adjustment. On the other hand, there exist system errors during the modeling and measurement process. For both reasons, stall happens frequently, resulting in high delay.

(ii) *BFA*: An essential difference between live streaming and VoD streaming is that the contents of live streaming is produced just before it is played. So even if the stream bitrate in live streaming is far below the network throughput, the buffer occupancy cannot accumulate quickly as data that have not been generated cannot be buffered. The weak accumulation of buffer leads to a low bitrate. Due to this phenomenon, *BFA* results in low delay but poor video quality.

(iii) *Single*: It has to choose a single target bitrate that is the lowest common denominator among all the receivers, and thus the target bitrate is lower compared to other schemes, leading to lower quality. Due to the lower bitrate, however, the delay and the stall duration are smaller for receivers with higher uplink throughput.

(iv) *Fixed*: It ignores the receiver state and network fluctuation, which leads to high bitrate and severe stalling.

(v) *Janus*: It comprehensively considers the uplink bandwidth of the sender and selects the highest bitrate allowed for the downlink transmission of all receivers. To ensure the smooth playback for users under low throughput condition, it conservatively chooses a lower bitrate, resulting in lower delay and fewer stalls.

(vi) *Mesh*: It generates several streams for multiple receivers simultaneously, which causes the uplink to become a bottleneck and severely limits the increase of the bitrate. Therefore, the bitrate is very low. It also has lower delay and fewer stalls.

(vii) Our algorithm *MultiLive* combines the advantage of *NLP* and *BFA*. On the premise of guaranteeing fluency, it elevates users' video quality under the limited throughput to maximize the personalized QoE. As shown in the figure, it performs well on both datasets.

Fig. 11 and Fig. 12 present the QoE performance for both datasets. We can see that: (i) The overall QoE value of *NLP* is relatively low due to the low accuracy of prediction; (ii) Due to the lower bitrates and the hysteresis of feedback, *BFA* does not perform well either; (iii) Due to the worse overall network condition in the Commercial Dataset, compared to the Belgium 4G/LTE Dataset, the score of video quality falls

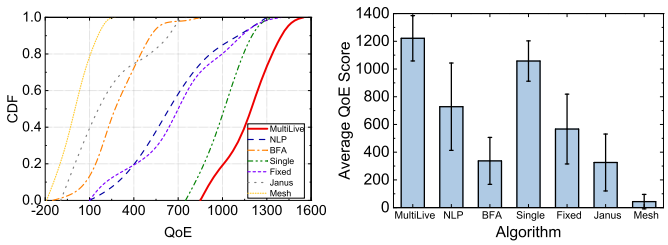


Fig. 11. QoE performance using the Commercial Dataset.

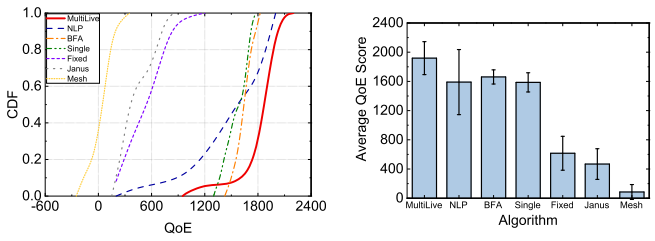


Fig. 12. QoE performance using the Belgium 4G/LTE Dataset.

quickly in the Commercial Dataset, according to the logarithmic characteristic of video quality function $q(\cdot)$, leading to a lower QoE for *BFA*; (iv) *Single* selects only one bitrate and the QoE effect is slightly worse; (v) *Fixed* selects the bitrate using initial network conditions, which change greatly during different time. So the QoE value is relatively unstable; (vi) Since *Janus* and *Mesh* has lower bitrate, which causes poor video quality, they both have lower QoE scores. Compared with *Fixed*, our proposed *MultiLive* algorithm improves QoE by 2-5$\times$.

F. Test Bed Evaluation

To evaluate the performance of our algorithm in a more realistic network scenario, we use a live streaming test bed to experimentally evaluate *MultiLive*, as shown in Fig. 13. The streamer uses commodity desktop with our specified implementation based on Chrome to connect to the server through a WiFi network. The desktop has an Intel 4-core i5-8250U CPU. The streamer utilizes its SVC software encoding capability to generate multi-rate video stream. To ensure the controllability of experiments, we deploy a network emulator [50] between the router and wireless access point (AP) to control network changes. We use a PC to configure network emulator, which can emulate a simple network link between physical network port pairs and does not classify the packets. All packets passing through the network emulator will be subjected to delay, dropping, or other impairments with some probability. We build three different network environments by controlling the packet loss rates and RTT: a *strong* network with packet loss rate less than 0.5% and RTT less than 10 ms, a *medium* network with packet loss rate between 0.5% and 2.5% and RTT less than 50 ms, and a *weak* network with packet loss rate more than 2.5% and RTT less than 100 ms. We measure the performance of *MultiLive* under these network conditions. The resulting average bitrate, average bitrate variation, stall ratio, and delay are shown in Fig. 14.

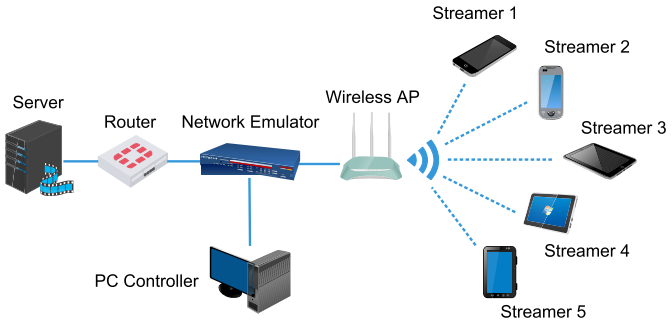


Fig. 13. Experimental test bed for strong, medium and weak network environment emulated with network emulator.

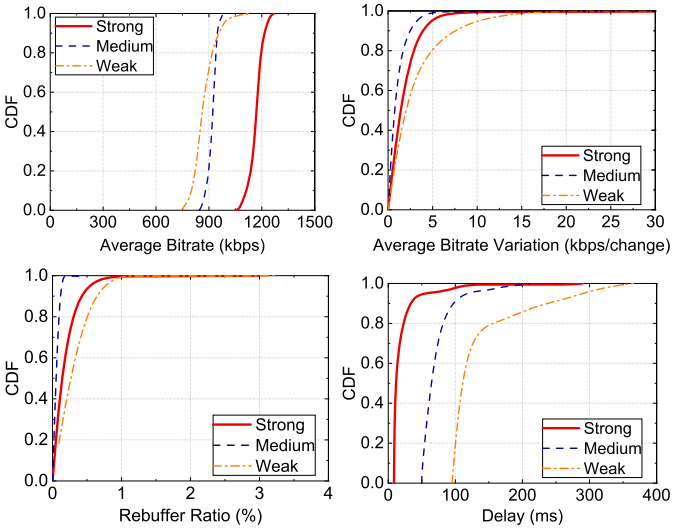


Fig. 14. Detailed performance for test bed.

As can be concluded from the figure, *MultiLive* increases the bitrate with the improvement of network condition. Meanwhile, it can control the delay within a low value which is slightly higher than RTT. Under strong network conditions, there is a 97% probability that the delay is less than 100 ms. For medium network, the probability is 90%. On the premise that the RTT under weak network occupies around 100 ms, its end-to-end delay is under 150 ms with the probability of 80%. Besides, the average bitrate variation and stall ratio are both satisfactory under different network conditions.

In addition, if we factor out the network propagation delay (RTT) component, which MultiLive cannot control, from the end-to-end delay, we obtain the average delay of 22 ms, 28 ms, and 55 ms for strong, medium, and weak network conditions respectively. This low delay points to the efficacy of MultiLive in supporting multi-party interactive live streaming in a network with low RTT.

G. Overhead Analysis

In this subsection, we perform some overhead analyses of the proposed system, including SVC encoding overhead, control packet overhead and algorithm execution time overhead. We conduct the experiments on a desktop, which has an Intel 4-core i5-8250U CPU.

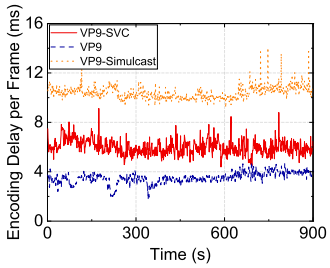


Fig. 15. Comparison of encoding delay per frame between VP9 and VP9-SVC encoders.

TABLE II
MEAN AND STANDARD DEVIATION OF TIME OVERHEAD FOR THREE ALGORITHM COMPONENTS

Component	Mean	Standard Deviation
NLP	47.31 ms	29.80 ms
BFA	0.82 ms	0.44 ms
Clustering	0.52 ms	0.11 ms

(1) To evaluate the encoding overhead of SVC, we use a Chrome WebRTC-internal tool to measure the encoding delay of VP9, VP9-SVC and VP9-Simulcast respectively in a typical video conference scenario. We configure VP9-SVC to use two spatial layers (spatial resolution: 640×480, 320×240) and three temporal layers (frame rate: 8, 15, 30 FPS). VP9-Simulcast produces multiple independent versions of the same stream with different resolutions. The result is shown in Fig. 15. We can find that in a typical video conference scenario, the VP9 encoder takes an average of 3.5 ms to encode a frame, while the VP9-SVC encoder requires an average of 6.0 ms per frame. Compared with the plain VP9, VP9-SVC only adds about 2.5 ms encoding delay per frame, which is acceptable. To achieve the same effect of SVC, VP9-Simulcast requires an average of 10.4 ms encoding delay per frame, which is higher than the delay in the case of using SVC. In addition, compared with the end-to-end 70 ms delay requirement [8], VP9-SVC only needs 6.0 ms encoding delay. Therefore, the encoding module is not the bottleneck in this scenario, but the network transmission is a bottleneck and needs to be optimized.

(2) The adaptive bitrate controller on the server takes, as inputs, the uplink/downlink throughput and buffer occupancy of each streamer to determine the bitrates to use. Therefore, the streamers need to periodically feed back the information using RTCP to the server. The frequency of feedbacks is related to the frequency of algorithm execution. For example, if the algorithm needs to be executed every 200 ms (as discussed in subsection B), the streamers need to feed back to the server at least every 200 ms. We make statistics on the throughput of RTP and RTCP packets when the system is running. The result is shown in Fig. 16. It can be seen that the throughput of data packets (RTP) is much greater than that of control packets (RTCP), and there is an order of magnitude difference. Consequently, compared with the video streaming data, control packet overhead is relatively small.

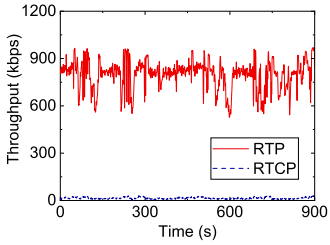
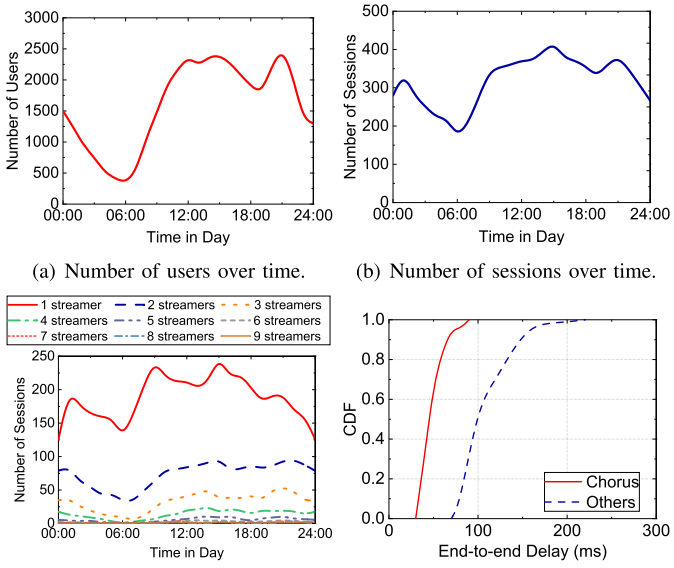


Fig. 16. Comparison of RTP and RTCP throughput.



(a) Number of users over time. (b) Number of sessions over time.

(c) Number of sessions participated by different number of streamers. (d) End-to-end delay.

Fig. 17. Some statistics for real-world deployment.

(3) Our algorithm includes three components: *NLP*, *BFA* and *Clustering* algorithm based on QoE loss. To better evaluate the performance of the algorithm, we measure the time overhead when each component is executed only separately. The result is shown in Table II. As can be seen from the table, the time overhead of *NLP* is within 100 ms. In other words, running *NLP* at a certain frequency does not have a high overhead. At present, we perform *NLP* every 2 seconds (as discussed in subsection B). The corresponding overhead caused by *NLP* is low. At the same time, the *BFA* and *Clustering* algorithm take less than 1 ms, which can fully support the requirements of frequent calls.

H. Real-World Deployment

We have deployed the algorithm on a commercial live streaming platform that provides such services for thousands of users. Our solution has been running on both commodity personal mobile devices and high-end special devices (including CPE and advanced desktops). Some statistics of the number of users and sessions over time are shown in Fig. 17. A session is a temporary and interactive information interchange between users. More than 2300 users and 400 sessions have been simultaneously online during some peak time. In our statistics, there are at most nine streamers participating in a session at the same time. As shown in the figure, as the number of streamers in a session increases, the number of sessions decreases. This may be because there is a constraint on the total bandwidth.

We also log the end-to-end delay experienced by the users. For applications of multiple streamers singing together which require a much tighter synchronization among the users, the end-to-end delay is generally 40–80 ms (need for better user access network); for other types of applications, such as ordinary video chat or video conference among streamers, the delay is generally 100–200 ms. They both use our algorithm. The result shows that our algorithm can well support real-world multi-party interactive live streaming system.

VIII. Related Work

Previous ABR algorithms: The previous ABR algorithms can be primarily grouped into four classes: rate-based, buffer-based, hybrid, and learning-based. (i) Rate-based methods estimate the available network throughput and request the next chunk at the highest bitrate that the network is predicted to support. For example, Akhtar *et al.* [10] proposed a system for automatically tuning ABR algorithm configurations in real time to match the current network state. Jiang *et al.* [37] presented a principled understanding of bitrate adaptation and analyzed several commercial players through the lens of an abstract player model. (ii) Buffer-based methods solely consider the client's buffer occupancy when deciding the bitrates for future chunks. For example, Huang *et al.* [51] considered using only the current buffer occupancy to pick a video rate, allowing for a simple function to map current buffer occupancy to video rate. Spiteri *et al.* [13] devised a buffer-based online control algorithm that uses Lyapunov optimization techniques to minimize rebuffering and maximize video quality. (iii) Hybrid methods use both throughput prediction and buffer occupancy to select bitrates that are expected to maximize QoE over several future chunks. For example, Yin *et al.* [14] proposed a novel model predictive control algorithm that can optimally combine throughput and buffer occupancy information. (iv) Learning-based methods use reinforcement learning to select bitrate adaptively. For example, Mao *et al.* [11] trained a neural network model that selects bitrates for future video chunks based on observations collected by client video players. Sengupta *et al.* [15] proposed a system which takes into consideration content preferences of users during adaptive video streaming over HTTP and employed the reinforcement learning model to enable optimal prefetch and bitrate decisions. In addition, some researchers have adopted other methods. For example, Yadav *et al.* [12] proposed a bitrate adaptation algorithm by modeling a client as an M/D/1/K queue. Lai *et al.* proposed a FoV-based bitrate adaptation algorithm for mobile virtual reality system to improve the QoE [52], [53]. However, all these methods only consider one streaming source and its delivery to a number of viewers in VoD streaming scenarios, rather than many-to-many in interactive live streaming scenarios. In addition, these methods select from discrete bitrate gears instead of adjusting on continuous bitrate domain.

Live streaming: Some previous studies provide architectures for live streaming delivery, mainly including two categories.

The first category is the centralized architecture. For example, Mukerjee *et al.* [54] provided real-time control over individual streams from the CDN side and employed centralized quality optimization for responsiveness. The second category is distributed architecture. For example, Liu *et al.* [55] designed an open P2P live video streaming system which can accommodate a variety of video coding schemes. However, they mainly consider one media source and its delivery to a number of viewers, which is quite different from us. There are also some studies that focus on crowdsourced live streaming, which generalizes the single-source streaming. Chen *et al.* [56] explored the emerging crowdsourced live streaming systems and designed cloud leasing strategy to optimize the cloud site allocation. They, however, do not account for real-time interaction, where the delay requirement is very harsh. In addition, He *et al.* [57] proposed a novel framework for crowdsourced livecast systems that offload the transcoding assignment to the massive viewers. Pang *et al.* [58] observed unique characteristics related to viewers (proactive and passive) and designed a deep neural network model to capture the viewer interaction pattern. Huang *et al.* [30] proposed a deep-learning based rate control algorithm for the real-time video streaming.

Multi-party video conferencing: Some previous studies have put forward multi-party cloud video conferencing architecture. Hu *et al.* [16] studied the server selection and server location optimization problem with a k-server mesh topology in distributed interactive video streaming applications to reduce end-to-end delay. Wu *et al.* [17] designed a fully decentralized algorithm to decide the best paths of streams and the most suitable surrogates along the paths. Hajiesmaili *et al.* [9] cast a joint problem of user-to-agent assignment and transcoding-agent selection, and proposed an adaptive parallel algorithm. Ooi and Van Renesse [59] divided up the in-network merging and transcoding process, and identified suitable cloud servers to run them, with the goal of minimizing the overall network cost. These studies, however, generally focus on the selection of transcoding server to minimize the cost of the service provider and the delay, and cannot meet the need of multi-party interactive live streaming where different online streamers may have different QoE preferences. Multi-party live streaming pays more attention to the user's experience and aims to improve the global QoE. Similar to our work, Amir *et al.* [60] proposed SCUBA, using scalable video coding and allowed the receiver to adjust the quality for different senders. But they do not consider global optimization that takes into consideration of QoE factors such as delay and smoothness. Small delay consideration is particular important for multi-party interactive live applications, where the tolerable delay is less than 70 ms [8], while for video conference, where participants take turns to talk, the tolerable delay is around 400 ms [61].

Bandwidth estimation and prediction: Some previous studies focus on optimizing bandwidth prediction process. For example, some studies have used time-series models. Jiang *et al.* [37] used the harmonic mean over the last 20 samples to predict throughput. Yin *et al.* [14] used the harmonic mean of the observed throughput of the last 5 chunks. Yadav *et al.* [12] compared different throughput prediction methods including Exponential Moving Average (EMA), Gradient Adaptive EMA, Low Pass EMA, and Kaufman's Adaptive Moving Average (KAMA). Some studies have used machine-learning models. Mirza *et al.* [38] used support vector regression for TCP throughput prediction. Xu *et al.* [39] used regression tree to forecast achievable network performance in real time. Other studies have used data-driven models. Sun *et al.* [40] used clusters of similar sessions and Hidden-Markov-Model (HMM) to achieve better throughput prediction. There are also some studies that probe available bandwidth. Mok *et al.* [21] employed the media data packets directly to make inline bandwidth measurement. Zhang *et al.* [20] used probing (with dummy traffic) to stabilizes adaptation. Li *et al.* [48] increased the sending rate additively and decreased it multiplicatively when congestion.

IX. Conclusion

In this paper, we propose an architecture for multi-party interactive live streaming. We build a QoE model and propose MultiLive, an adaptive bitrate control algorithm. Specifically, we apply non-linear programming to get the target bitrate for each pair of online streamers, and adjust the bitrate according to the buffer feedback to avoid the accumulation of system errors. To alleviate the problem of limited uplink transmission rate, we use bitrate clustering to reduce the number of streams to transmit from a streamer. We also propose an available bandwidth measurement algorithm to passively probe the available bandwidth to improve the effectiveness of MultiLive. Our results from extensive trace-driven simulations and test bed experiments demonstrate that MultiLive outperforms the fixed bitrate algorithm, with 2-5$\times$ improvement of the average QoE. Furthermore, we deploy the algorithm on a commercial live streaming platform that provides such services for more than 2300 users. The end-to-end delay has been reduced to around 100 ms, which is much lower than 400 ms used as the delay threshold in existing schemes for video conferencing.

References

[1] Z. Lu, H. Xia, S. Heo, and D. Wigdor, "You watch, you give, and you engage: A study of live streaming practices in China," in *Proc. ACM CHI*, 2018, pp. 1–13.
[2] O. L. Haimson and J. C. Tang, "What makes live events engaging on Facebook Live, Periscope, and Snapchat," in *Proc. ACM CHI*, 2017, pp. 48–60.
[3] J. C. Tang, G. Venolia, and K. M. Inkpen, "Meerkat and Periscope: I stream, you stream, apps stream for live streams," in *Proc. ACM CHI*, 2016, pp. 4770–4780.
[4] L. Provensi, A. Singh, F. Eliassen, and R. Vitenberg, "Maelstream: Self-organizing media streaming for Many-to-Many interaction," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1342–1356, Jun. 2018.
[5] F. Wang, J. Liu, M. Chen, and H. Wang, "Migration towards cloud-assisted live media streaming," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 272–282, Feb. 2016.
[6] *Inke*. Accessed: Nov. 22, 2021. [Online]. Available: https://www.inke.com
[7] *Douyu*. Accessed: Nov. 22, 2021. [Online]. Available: https://www.douyu.com
[8] N. Schuett, "The effects of latency on ensemble performance," Bachelor Thesis, Dept. Center Comput. Res. Music Acoust., Stanford Univ., Stanford, CA, USA, 2002.
[9] M. H. Hajiesmaili, L. T. Mak, Z. Wang, C. Wu, M. Chen, and A. Khonsari, "Cost-effective low-delay design for multiparty cloud video conferencing," *IEEE Trans. Multimedia*, vol. 19, no. 12, pp. 2760–2774, Jun. 2017.
[10] Z. Akhtar *et al.*, "Oboe: Auto-tuning video ABR algorithms to network conditions," in *Proc. ACM SIGCOMM*, 2018, pp. 44–58.

[11] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proc. ACM SIGCOMM*, 2017, pp. 197–210.
[12] P. K. Yadav, A. Shafiei, and W. T. Ooi, "QUETRA: A queuing theory approach to DASH rate adaptation," in *Proc. ACM MM*, 2017, pp. 1130–1138.
[13] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "BOLA: Near-optimal bitrate adaptation for online videos," in *Proc. 35th Annu. Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.
[14] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over HTTP," in *Proc. ACM SIGCOMM*, 2015, pp. 325–338.
[15] S. Sengupta, N. Ganguly, S. Chakraborty, and P. De, "HotDASH: Hotspot aware adaptive video streaming using deep reinforcement learning," in *Proc. IEEE ICNP*, Sep. 2018, pp. 165–175.
[16] Y. Hu, D. Niu, and Z. Li, "A geometric approach to server selection for interactive video streaming," *IEEE Trans. Multimedia*, vol. 18, no. 5, pp. 840–851, May 2016.
[17] Y. Wu, C. Wu, B. Li, and F. C. M. Lau, "VSkyConf: Cloud-assisted multi-party mobile video conferencing," in *Proc. 2nd ACM SIGCOMM Workshop Mobile Cloud Comput.*, 2013, pp. 33–38.
[18] Y. Guo, Q. Yang, J. Liu, and K. S. Kwak, "Quality-aware streaming in heterogeneous wireless networks," *IEEE Trans. Wireless Commun.*, vol. 16, no. 12, pp. 8162–8174, Dec. 2017.
[19] C. Dong, W. Wen, T. Xu, and X. Yang, "Joint optimization of data-center selection and video-streaming distribution for crowdsourced live streaming in a geo-distributed cloud platform," *IEEE Trans. Netw. Service Manage.*, vol. 16, no. 2, pp. 729–742, Jun. 2019.
[20] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "AWStream: Adaptive wide-area streaming analytics," in *Proc. ACM SIGCOMM*, 2018, pp. 236–252.
[21] R. K. Mok, X. Luo, E. W. Chan, and R. K. Chang, "QDASH: A QoE-aware DASH system," in *ACM MMSys*, 2012, pp. 11–12.
[22] J. van der Hooft *et al.*, "HTTP/2-based adaptive streaming of HEVC video over 4G/LTE networks," *IEEE Commun. Lett.*, vol. 20, no. 11, pp. 2177–2180, Nov. 2016.
[23] *Hardware Recommendations*. Accessed: Nov. 22, 2021. [Online]. Available: https://www.twitch.tv/creatorcamp/en/setting-up-your-stream/hardware-recommendations
[24] A. Eleftheriadis, "SVC and video communications," Vidyo, Hackensack, NJ, USA, White Paper, 2011.
[25] G. Bakar, R. A. Kirmizioglu, and A. M. Tekalp, "Motion-based rate adaptation in webRTC videoconferencing using scalable video coding," *IEEE Trans. Multimedia*, vol. 21, no. 2, pp. 429–441, Feb. 2019.
[26] G. Zhang and J. Y. Lee, "LAPAS: Latency-aware playback-adaptive streaming," in *Proc. WCNC*, Apr. 2019, pp. 1–6.
[27] L. Sun, T. Zong, S. Wang, Y. Liu, and Y. Wang, "Tightrope walking in low-latency live streaming: Optimal joint adaptation of video rate and playback speed," in *Proc. MMSys*, 2021, pp. 200–213.
[28] A. Ahmed, Z. Shafiq, H. Bedi, and A. Khakpour, "Suffering from buffering? Detecting QoE impairments in live video streams," in *Proc. ICNP*, Oct. 2017, pp. 1–10.
[29] X. Zuo, Y. Cui, M. Wang, T. Xiao, and X. Wang, "Low-latency networking: Architecture, techniques, and opportunities," *IEEE Internet Comput.*, vol. 22, no. 5, pp. 56–63, Sep. 2018.
[30] T. Huang, R.-X. Zhang, C. Zhou, and L. Sun, "QARC: Video quality aware rate control for real-time video streaming based on deep reinforcement learning," in *Proc. MM*, 2018, pp. 1208–1216.
[31] M. Mu *et al.*, "A scalable user fairness model for adaptive video streaming over SDN-assisted future networks," *IEEE J. Sel. Areas Commun.*, vol. 34, no. 8, pp. 2168–2184, Aug. 2016.
[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press, 2009.
[33] J. Huang, T. M. Smith, G. M. Henry, and R. A. Van De Geijn, "Strassen's algorithm reloaded," in *Proc. SC*, 2016, pp. 690–701.
[34] W. Huang, Y. Zhou, X. Xie, D. Wu, M. Chen, and E. Ngai, "Buffer state is enough: Simplifying the design of QoE-aware HTTP adaptive video streaming," *IEEE Trans. Broadcast.*, vol. 64, no. 2, pp. 590–601, Jun. 2018.
[35] Y. Qin *et al.*, "A control theoretic approach to ABR video streaming: A fresh look at PID-based rate adaptation," *IEEE Trans. Mobile Comput.*, vol. 19, no. 11, pp. 2505–2519, Nov. 2020.
[36] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-means clustering algorithm," *Appl. Statist.*, vol. 28, no. 1, p. 100, 1979.
[37] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE," in *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol.*, 2014, pp. 97–108.
[38] M. Mirza, J. Sommers, P. Barford, and X. Zhu, "A machine learning approach to TCP throughput prediction," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 97–108, Jun. 2007.
[39] Q. Xu, S. Mehrotra, Z. Mao, and J. Li, "PROTEUS: Network performance forecast for real-time, interactive mobile applications," in *Proc. ACM MobiSys*, 2013, pp. 347–350.
[40] Y. Sun *et al.*, "CS2P: Improving video bitrate selection and adaptation with data-driven throughput prediction," in *Proc. ACM SIGCOMM*, 2016, pp. 272–285.
[41] J. F. Kurose and K. W. Ross, *Computing Networking: A Top-Down Approach*. Reading, MA, USA: Addison-Wesley, 2010.
[42] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano, "Janus: A general purpose webRTC gateway," in *Proc. ACM IPTComm*, 2014, pp. 1–8.
[43] *Janus Gateway*. Accessed: Nov. 22, 2021. [Online]. Available: https://github.com/meetecho/janus-gateway
[44] T. Reddy, P. Patil, D. Wing, and B. Ver Steeg, "WebRTC UDP firewall traversal," in *Proc. IAB Workshop Stack Evol. Middlebox Internet (SEMI)*, 2015, pp. 1–5.
[45] J. Lazzaro, "RFC 4571: Framing real-time transport protocol (RTP) and RTP control protocol (RTCP) packets over connection-oriented transport," Internet Eng. Task Force, Tech. Rep., 2006. [Online]. Available: https://datatracker.ietf.org/doc/rfc4571/
[46] L. Miniero, *HTTP Fallback for RTP Media Streams*, document Internet-Draft draft-miniero-rtcweb-http-fallback-00, IETF, 2012.
[47] *TC: Traffic Control*. Accessed: Nov. 22, 2021. [Online]. Available: https://linux.die.net/man/8/tc
[48] Z. Li *et al.*, "Probe and adapt: Rate adaptation for HTTP video streaming at scale," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 4, pp. 719–733, Apr. 2014.
[49] *Mesh Topology*. Accessed: Nov. 22, 2021. [Online]. Available: https://github.com/rtc-io/rtc-mesh
[50] *Network Emulator*. Accessed: Nov. 22, 2021. [Online]. Available: http://www.msytest.cn
[51] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," in *ACM SIGCOMM*, 2014, pp. 184–198.
[52] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, N. Dai, and H.-S. Lee, "Furion: Engineering high-quality immersive virtual reality on Today's mobile devices," *IEEE Trans. Mobile Comput.*, vol. 19, no. 7, pp. 1586–1602, Jul. 2020.
[53] Z. Lai, Y. Cui, Z. Wang, and X. Hu, "Immersion on the edge: A cooperative framework for mobile immersive computing," in *ACM SIGCOMM Posters Demos*, 2018, pp. 39–41.
[54] M. K. Mukerjee, D. Naylor, J. Jiang, D. Han, S. Seshan, and H. Zhang, "Practical, real-time centralized control for CDN-based live video delivery," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 311–324, Sep. 2015.
[55] Z. Liu, Y. Shen, K. W. Ross, S. S. Panwar, and Y. Wang, "Substream trading: Towards an open P2P live streaming system," in *Proc. ICNP*, Oct. 2008, pp. 94–103.
[56] F. Chen, C. Zhang, F. Wang, and J. Liu, "Crowdsourced live streaming over the cloud," in *Proc. IEEE INFOCOM*, 2015, pp. 2524–2532.
[57] Q. He, C. Zhang, and J. Liu, "CrowdTranscoding: Online video transcoding with massive viewers," *IEEE Trans. Multimedia*, vol. 19, no. 6, pp. 1365–1375, Jun. 2017.
[58] H. Pang *et al.*, "Optimizing personalized interaction experience in crowd-interactive livecast: A cloud-edge approach," in *Proc. ACM MM*, 2018, pp. 1217–1225.
[59] W. T. Ooi and R. Van Renesse, "Distributing media transformation over multiple media gateways," in *ACM MM*, 2001, pp. 159–168.
[60] E. Amir, S. McCanne, and R. Katz, "Receiver-driven bandwidth adaptation for light-weight sessions," in *Proc. MM*, 1997, pp. 415–426.
[61] *One-Way Transmission Time, Series G: Transmission Systems and Media, Digital Systems and Networks*, document G. 114, Telecommunication Standardization Sector, ITU-T, 2000.

Ziyi Wang received the B.E. degree in networking engineering from the Dalian University of Technology, Liaoning, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include video streaming and mobile computing.

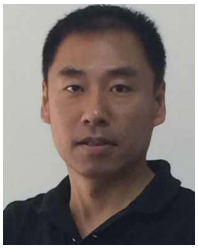
Yong Cui (Member, IEEE) received the B.E. and Ph.D. degrees in computer science and engineering from Tsinghua University, China, in 1999 and 2004, respectively. He is currently a Full Professor with the Computer Science Department, Tsinghua University. His major research interests include mobile cloud computing and network architecture.

Wei Tsang Ooi (Member, IEEE) received the Ph.D. degree in computer science from Cornell University, Ithaca, NY, USA. He is currently an Associate Professor with the Department of Computer Science, National University of Singapore, where he does research in multimedia systems, distributed systems, and computer networking.

Xiaoyu Hu received the B.E. degree from Zhejiang University, Zhejiang, China, in 2014. He is currently pursuing the master's degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include video streaming and mobile computing.

Zhen Cao received the Ph.D. degree from Peking University, China. He is currently a Principle Engineer with Huawei Technologies Company Ltd., China. His research interests include sensor networks, security and privacy, cellular network technologies, IPv6, and SDN/NFV.

Xin Wang (Member, IEEE) received the B.S. and M.S. degrees from the Beijing University of Posts and Telecommunications, China, and the Ph.D. degree from Columbia University, USA. She is currently an Associate Professor with the Department of Electrical and Computer Engineering, State University of New York at Stony Brook. Her research interests include algorithm and protocol design in wireless networks, and mobile and distributed computing.



Yi Li received the B.E. and Ph.D. degrees in computer science and engineering from Tsinghua University in 1996 and 2000, respectively. He is currently with Beijing Powerinfo Company Ltd., as a CTO. His research interests include video encoding and transmission.