New Approaches to Improving Live Video Delivery over Content Delivery Networks

by

Huan Wang
B.Eng., Southwest Jiaotong University, 2013
M.Eng., University of Electronic Science and Technology of China, 2016

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

New Approaches to Improving Live Video Delivery over Content Delivery Networks

by

Huan Wang
B.Eng., Southwest Jiaotong University, 2013
M.Eng., University of Electronic Science and Technology of China, 2016

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Sudhakar Ganti, Departmental Member
(Department of Computer Science)

Dr. Xiaodai Dong, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Sudhakar Ganti, Departmental Member
(Department of Computer Science)

Dr. Xiaodai Dong, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT

The live streaming services have gained extreme popularity in recent years. The spiky traffic, as well as the real-time property of live videos, make it challenging for content delivery networks (CDNs) to guarantee the Quality-of-Experiences (QoE) of viewers. The caching and delivery mechanism of live videos over the current CDN architecture has its limitation since the current CDNs are not designed for live videos in the first place. As a result, it may lead to deteriorated QoE to the end viewers such as long startup latency. In this dissertation, with the help of CDN edge servers, we focus on the QoE improvement solutions of three problems in live video delivery, covering the research fields of i) multi-CDN content delivery, ii) QoE optimization of HTTP-based live video delivery, and iii) live video replication over edge CDN servers.

First, in order to improve the content delivery performance under current multi-CDN strategies, we propose a feasible and efficient solution to multi-CDN, termed as CDN semi-federation. Compared with a full *multi-CDN federation*, CDN semi-federation can better schedule and utilize the resources from multiple CDNs without requiring full CDN interconnection (CDNI), which poses significant technical obstacles not easy to solve in the short term. The semi-federation model requires an authoritative and trusted third-party consortium formed by *voluntary* CDN vendors,

who need to disclose their dynamic information (e.g., PoP footprints and service capabilities) to the consortium. The authoritative consortium adopts a centralized control to provide traffic delivery guidance to CDNs by leveraging the resources from multiple CDNs and reshaping the traffic demand assigned to each CDN. Compared with CDNI, CDN semi-federation: i) releases the CDN vendors from the complex technical and business obstacles of interconnecting with multiple CDNs, and ii) avoids the sub-optimal content delivery decisions made by distributed CDNs.

Second, to optimize the QoE of HTTP-based live video delivery over CDN edge servers, we propose a reinforcement learning-based dynamic *IVS* selection scheme (*Rldish*) deployed on edge CDN server to dynamically select a suitable initial video segment (IVS) of a live streaming. *Rldish* uses a real-time exploration and exploitation (E2) model to learn the *IVS* selection automatically, and is deployed as a virtual network function (VNF) on the CDN edge server by the CDN operator. *Rldish* makes the *IVS* decisions on a *per-stream basis* to avoid high overhead in per-user based throughput estimation. Since an edge CDN server generally serves its proximal end users, viewers accessing the same live video usually share the common video delivery path from the origin server to the edge and generally experience the similar network conditions when fetching the same video from the edge. Based on this observation, *Rldish* continuously updates the currently optimal decisions on *IVS* selection for the live viewers on a *per-stream basis*, based on the real-time QoE measurements and feedback. The decisions are then updated into the media playlist files of each stream for the subsequent live viewers.

Third, to solve the cache miss problem in edge-assisted live video delivery, we propose a proactive live video edge replication scheme (*PLVER*). *PLVER* first conducts a *one-to-multiple* stable allocation between edge clusters and user groups to balance the load of live requests over edge servers. In this way, each user group is assigned to its most preferred edge cluster whenever possible. Based on the allocation result, *PLVER* then proposes an efficient proactive live video edge replication (push) algorithm to speed up the edge replication process by using real-time statistical viewership of the user groups allocated to a cluster. We conduct extensive trace-driven evaluations, covering 0.3 million Twitch viewers and more than 300 Twitch channels. The results demonstrate that with *PLVER*, edge servers can carry 28% and 82% more traffic than the auction-based replication method and the caching on requested time method, respectively.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

my supervisor, Dr. Kui Wu, for giving me the best mentoring and support during my whole Ph.D. time. Whenever I am stuck with my research, you are always there to help. Your encouragement and patience are much appreciated. I would like to express my deep and sincere gratitude to you.

Dr. Jianping Wang, for hosting me and mentoring me during my visiting to City University of Hong Kong for research collaborations.

my parents, for their unconditional love and support over my entire life.

my friends and collaborators in UVic and CityU HK, for the valuable advice and the best moments we have spent together.

my committee members, Dr. Sudhakar Ganti and Dr. Xiaodai Dong, for the precious comments.

DEDICATION

To my dear parents Dejuan and Hu.

Chapter 1

Introduction

In this chapter, we describe the motivation for our research efforts to improve the quality of experience (QoE) of live video viewers over the content delivery networks, and explain our research goals and contributions.

1.1 Motivation

A content delivery network (CDN) is a globally distributed network of proxy servers deployed at the network edge such that end users (EUs) can access the Internet content with low latency and high Quality of Experience (QoE). Although live streaming services have been gaining increasing popularity in recent years [1], CDNs continue to struggle with delivering high-quality live videos to viewers while guaranteeing their Quality-of-Experiences (QoE) [2, 3]. To address such an issue, CDN operators rely on the widely distributed edge servers (e.g., the edge data centers [4]) to handle the increasing demand of live videos. Using edge servers as the cache, most viewers can fetch the requested contents directly from the edge cache rather than the original streaming servers, thus alleviate the traffic burden of the origin servers.

Fig. 1.1 shows the global view of the live video delivered via the edge servers over the Internet, where each live video stream is encoded and split into a sequence of small video segments. In order to watch a live stream, the clients could download the segments sequentially by sending the `HTTP GET` requests. The requests from the client will first be dispatched to the local CDN edge servers (PoPs), where if the requested video segment is already cached in the edge servers, then the requests could get the response there directly. Otherwise, the edge servers will issue a new request to the

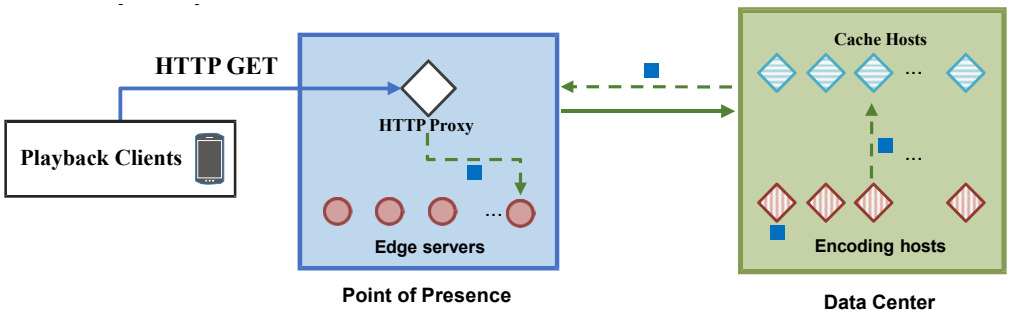


Figure 1.1: The global view of live video delivery over the Internet

data center to fetch the requested video segment. However, due to the unique features of live videos and edge server (e.g., the real-time properties of live video and limited service capacities of edge servers), it is extremely difficult to guarantee the QoE of live video viewers. A bad QoE implies a high abandonment rate of users for the content providers. Therefore, aiming at improving the QoE of live viewers as well as the performance of live video delivery, we try to address three critical research obstacles that limit the performance of live video delivery.

First, how to improve the performance of Multi-CDN content delivery? The booming market of CDNs offers a great opportunity for content providers (CPs) to shop around among multiple CDNs. Many CPs (e.g., the live video providers) now use multiple CDNs to cache and deliver their content such that they can dynamically select the CDNs with better QoE, based on certain criteria such as geographic locations of end users [5, 6]. In some cases, CPs rely on a CDN broker to deliver content over multiple CDNs [7]. The CDN broker can release CPs from the task of selecting CDN services, thus allow them to focus more on their own content business. We call the above multi-CDN solution *content multihoming*. Content multihoming, while widely adopted by CPs, may not benefit CDNs. The main reason is that CPs or CDN brokers make content delivery decisions based on their local measurement of network conditions and capacities. Measurement studies indicate that the CDN selection for content multihoming is largely based on proximity and latency [8] or statically configured [9]. Since CPs and brokers have a limited view and information on CDNs, it is difficult for them to make the globally optimal scheduling for content delivery. To make matters worse, CDN vendors, especially those small ones having limited CDN Point-of-Presences (PoPs), now receive fewer exclusive contracts because CPs can shop around over multiple CDNs.

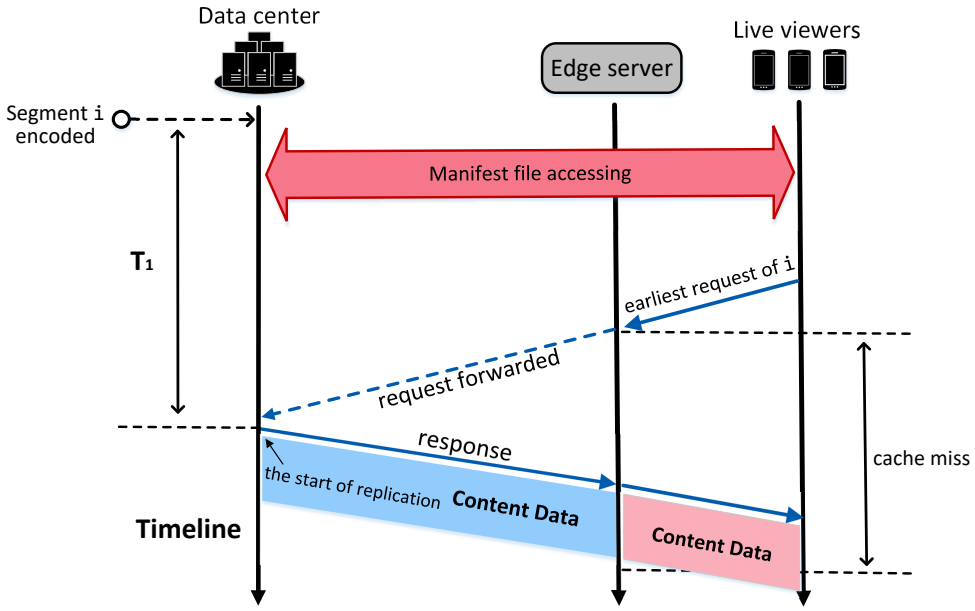


Figure 1.2: Client-driven content caching (replication) for live videos.

Second, how to improve the performance of live video delivery utilizing the unique features of live videos? Compared with regular videos, live videos generally have quite spiky traffic, which means the viewer popularity of live streams usually grows and drops very rapidly [10]. In particular, live videos often encounter the "thundering herd" problem [11, 12]: a large number of users, sometimes on the scale of millions, may start to watch the same live video simultaneously when some popular events or online celebrities start a live broadcast. Besides, live video delivery nowadays has stringent latency requirements due to the new breed of live video services that support interactive live video streaming. These services allow the broadcasters to interact with their stream viewers in real-time during the streaming process. In order to support the high interactivity, it requires low-latency end-to-end delivery while maintaining the Quality of Experience (QoE) for live viewers [13, 14, 15]. One efficient way to solve the thundering herd problem while maintaining low latency in live videos is to utilize edge caches. For example, Facebook uses edge PoPs distributed worldwide to deliver their live traffic [11]. Providing contents via the edge (e.g., CDN edge servers, crowdsourced edge devices [16]) makes contents much closer to the end users and alleviates the traffic burden of backbone networks to the cloud.

Third, how to solve the cache miss problem within the live video delivery? When

a large number of end users request for a newly generated video segment at the same time, this segment may not have enough time to be cached in the edge caches due to the real-time property of live streaming [17, 18]. The edge server would return a cache miss for the first group of requests that arrive at the edge before the segment is fully cached. These cache-missed requests would pass the edge cache and go all the way to the origin server. As a result, it would lead to deteriorated QoE to the live viewers (e.g., increased startup latency and playback stall rates). According to Facebook [11], around 1.8% of their Facebook Live requests encountered cache miss at the edge layer, and caused failures at the origin server level. Note that 1.8% is a significant number considering the large number of total live viewers. To make matters worse, high revolution videos (e.g., virtual reality (VR) streams) need more time to be replicated to the edge and would create an even higher cache miss rate.

1.2 Research Objectives and Contributions

Tackling the aforementioned challenges in live video delivery, we focus on three critical problems in this dissertation: i) multi-CDN content delivery, ii) QoE optimization of HTTP Live Streaming (HLS), and iii) live video replication over edge CDN servers.

1.2.1 Speeding up Multi-CDN Content Delivery

In order to build a better content delivery ecosystem for both CDN vendors and CPs, the concept of *multi-CDN federation* has been proposed recently as a promising business model [19, 20], where standalone CDNs are interconnected such that their collective PoPs and resources can be leveraged for end-to-end content delivery [21]. For CDN vendors (especially small CDNs with limited PoPs and resources), through the extended footprints and the leveraged resources, CDN federation can provide better QoE (e.g., lower latency) to end users and reduce the cost of redundantly deploying CDN PoPs. For CPs, CDN federation reduces the tedious contract and negotiation work between a CP and multiple CDNs. It also releases CPs from the technical difficulties of monitoring multiple CDNs and dynamically selecting the right ones to optimize their content delivery. It is expected that multi-CDN federation would attract more customers and bring a triple-win situation for the CDN vendor, the CP, and the end users.

Nevertheless, to form a full multi-CDN federation, CDN Interconnection (CDNI) [22,

23, 24] is required for dynamic traffic exchange among federated CDNs. CDNI requires a set of newly built interfaces and mechanisms to interconnect multiple CDNs such that the downstream CDNs (dCDN) is able to deliver content on behalf of the upstream CDN (uCDN). The interconnected CDNs not only need protocols and interfaces to exchange dynamic information (e.g., footprints and capabilities), but also need to replicate content from uCDN to dCDN. All these pose technical barriers. Furthermore, the optimization of content delivery in CDNI has also not been well researched.

To improve the content delivery performance over the current multi-CDN architecture, we propose a *CDN semi-federation* solution where the federated CDNs are independent of each other. The CDN semi-federation model requires an authoritative and trusted third-party consortium formed by *voluntary* CDN vendors, which need to disclose their dynamic information (e.g., PoP footprints and service capabilities) to the consortium. The authoritative consortium adopts a centralized control to provide traffic delivery guidance to CDNs by leveraging the resources from multiple CDNs and reshaping the traffic demand assigned to each CDN. Compared with CDNI, CDN semi-federation: i) releases the CDN vendors from the complex technical and business obstacles of interconnecting with multiple dCDNs, and ii) avoids the sub-optimal content delivery decisions made by distributed CDNs, because CDNs will obey the centralized delivery guidance from the consortium instead. This contribution has been published in [25].

1.2.2 Edge-Assisted QoE Optimization of HTTP-based Live Video Delivery

Recent years have seen a rapidly increasing traffic demand for *HTTP-based* high-quality live video streaming (e.g., HLS [26] and MPEG-DASH [27]). The surging traffic demand, as well as the real-time property of live videos, make it challenging for content delivery networks (CDNs) to guarantee the Quality-of-Experiences (QoE) of viewers. The initial video segment (*IVS*) of live streaming plays an important role in the QoE of live viewers, particularly when users require fast join time and smooth view experience. Existing *IVS* selection strategies either use a fixed value [28] or use the "optimal" value [18]. In the former, the RFC standard of HTTP Live Streaming (HLS) suggests that "client should not choose a segment that starts within *three* segment durations (the maximum playback duration of video segments in the playlist)

from the end of the playlist file" in order to avoid playback stalls [28]. In the latter, the "optimal" *IVS* value is derived to match the current network conditions (e.g., throughput) [18].

Clearly, the former will not work well for high-quality live video streaming due to the dynamic network conditions. The latter is promising but has two main pitfalls. First, it relies on the *per-user* based network throughput estimation. The network throughput is related to multiple complex factors (e.g., RTT and router buffer size), which frequently changes over time [29]. This can incur high computational overhead, especially for live videos where the number of live viewers is large [11]. Second, when a user joins a live channel, the server can only infer the user's network throughput through the signal strength (e.g., RSRP, RSRQ and RSSI in LTE) and mobility pattern (e.g., fast, slow, static). The network throughput estimation in this case may not be accurate, leading to a suboptimal choice of *IVS*. In practice, the overhead of searching for the "optimal" value may offset the benefit. It is also hard to quickly react to the network condition change.

To overcome the above problem, we propose a reinforcement learning-based dynamic *IVS* selection scheme (*Rldish*) deployed on edge CDN server to maintain a balance between exploring suboptimal decisions and exploiting currently optimal decisions. *Rldish* uses a real-time *exploration and exploitation (E2) model [30] to learn the IVS selection automatically, and is deployed as a virtualized network function (VNF) on the CDN edge server by the CDN operator. It can work seamlessly with existing edge CDN proxy (cache) server (e.g., Nginx) [11, 31], and can also react to the network condition (throughput) change via real-time exploration.*

Rldish makes the *IVS* decisions on a *per-stream basis* to avoid high overhead in per-user based throughput estimation. Since an edge CDN server generally serves its proximal end users, viewers accessing the same live video usually share the common video delivery path from the origin server to the edge and generally experience the similar network conditions when fetching the same video from the edge. Based on this observation, *Rldish* continuously updates the currently optimal decisions on *IVS* selection for the live viewers on a *per-stream basis*, based on the real-time QoE measurements and feedback. The decisions will then be updated into the media playlist files of each stream for the subsequent live viewers. This contribution has been published in [32].

1.2.3 Live Video Replication over Edge CDN Servers

Due to the spiky traffic patterns and the stringent requirement on latency of live videos, live streaming service providers nowadays more rely on the edge caches distributed all over the world to deliver their live traffic [11]. Nevertheless, when applying edge-assisted live video delivery, there exists a cache miss problem: when a large number of end users request for a newly generated video segment at the same time, this segment may not has enough time to be cached in the edge caches due to the real-time property of live streaming [17, 18]. The edge server would return a cache miss for the first group of requests that arrive at the edge before the segment is fully cached.

The root cause of the cache miss problem is mainly because the current client-driven caching strategy was not designed for live videos in the first place. Since caching process in the current content delivery networks (CDNs) is normally triggered by the client requests, the video segments caching (replication) will only commence when the cloud responds to the first request for a video segment. While this strategy makes sense when delivering regular content, it slows down the caching process in the context of live videos: there exists a time *gap* between the time when a segment is generated from the cloud and when the caching process starts. This gap mainly consists of two parts: i) the time that the the playback clients obtain the availability information of the newly encoded video segments, and ii) the time it takes for the clients to send their first segment request. However, in the current pull-based CDN architecture, both times are difficult to narrow down. This motivates us to rethink the caching design of live video delivery. Can the cloud CDN server adopts a video push model to proactively replicate the newly encoded video segments into the appropriate edge servers?

Based on the above motivation, we propose a proactive live video edge replication scheme (*PLVER*) to resolve the cache miss problem in live video delivery. *PLVER* first conducts a *one-to-multiple* stable allocation between edge clusters and user groups to balance the load of live requests over edge servers. In this way, each user group is assigned to its most preferred edge cluster whenever possible. Based on the allocation result, *PLVER* then proposes an efficient proactive live video edge replication (push) algorithm to speed up the edge replication process by using real-time statistical viewership of the user groups allocated to the cluster. We perform comprehensive experiments to evaluate the performance of *PLVER*. Trace-driven allocations between

641 edge clusters and 1253 user groups are conducted, covering 64 ISP providers and 470 cities. Based on the allocation results, we further evaluate the performance of the video replication algorithm using traces of 0.3 million Twitch viewers and more than 300 Twitch channels. Performance results demonstrate the superiority of *PLVER*.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows:

In Chapter 2, we propose a CDN semi-federation architecture as an alternative multi-CDN solution, which can be easily implemented over existing CDN infrastructures.

In Chapter 3, in order to improve the QoE of HTTP-based live video streaming, we propose a reinforcement learning based dynamic IVS selection scheme deployed on edge CDN servers to learn the IVS selection on a per-stream basis.

In Chapter 4, we propose a proactive live video push scheme to resolve the cache miss problem existed in live video delivery. It first conducts a one-to-multiple stable allocation between edge clusters and user groups, then adopts a proactive video replication algorithm to speed up the live video replication process among the edge servers.

In Chapter 5, we conclude the dissertation and propose future research in relevant research fields.

Chapter 2

Speeding up Multi-CDN Content Delivery

In this chapter, we will propose a feasible and efficient solution to multi-CDN content delivery, termed as CDN *semi-federation*, which can better schedule and utilize the resources from multiple CDNs. Live video providers are expected to benefit from the improved QoE of their live viewers (e.g., lower latency) by using our multi-CDN strategy.

2.1 Introduction

The booming market of CDNs offers a great opportunity for content providers (CPs) to shop around among multiple CDNs. Many CPs now use multiple CDNs to cache and deliver their content such that they can dynamically select the CDNs with better performance, based on the criteria such as geographic locations of end users [5, 6]. In some cases, CPs rely on a CDN broker to deliver content over multiple CDNs [7]. The CDN broker can release CPs from the task of selecting CDN services, thus allow them to focus more on their own content business. We call the above multi-CDN solution *content multihoming*.

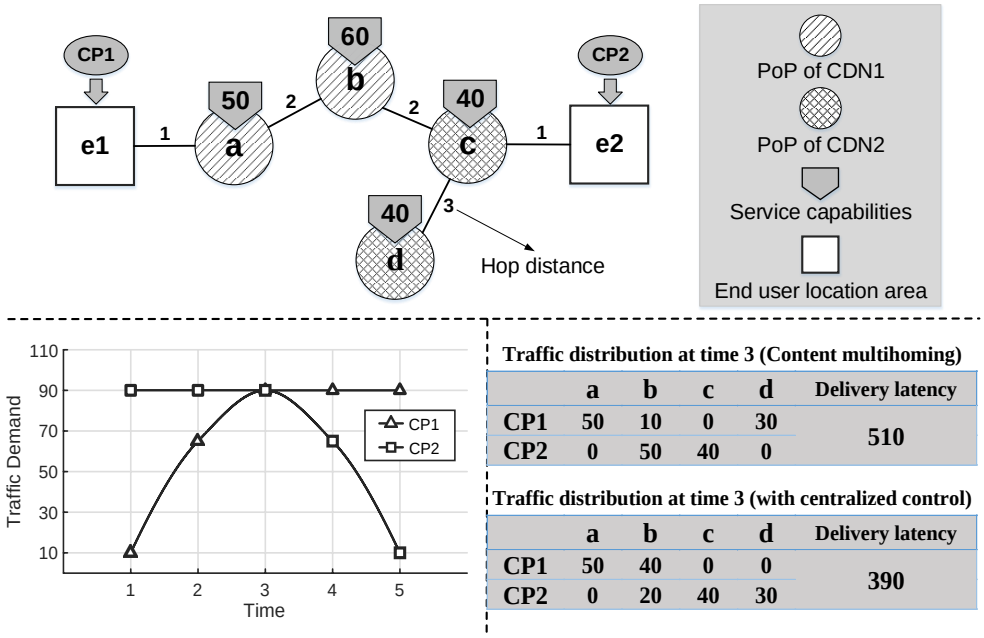
Content multihoming, while widely adopted by CPs, may not benefit CDNs, in terms of both performance and revenue. The main reason is that CPs or CDN brokers make content delivery decisions based on their local measurement of network conditions and capacities. Measurement studies indicate that the CDN selection for content multihoming is largely based on proximity and latency [8] or statically

configured [9]. Since CPs and brokers have a limited view and information on CDNs, it is difficult for them to make the globally optimal scheduling for content delivery. To make matters worse, CDN vendors, especially those small ones having limited CDN Point-of-Presences (PoPs), now receive fewer exclusive contracts because CPs can shop around over multiple CDNs.

To build a better content delivery ecosystem for both CDN vendors and CPs, *multi-CDN federation* has been proposed as a promising business model [19, 20], where standalone CDNs are interconnected such that their collective PoPs and resources can be leveraged for end-to-end content delivery [21]. For CDN vendors (especially small CDNs with limited PoPs and resources), through the extended footprints and the leveraged resources, CDN federation can provide better QoE (e.g., lower latency) to end users and reduce the cost of redundantly deploying CDN PoPs. For CPs, CDN federation reduces the tedious contract and negotiation work between a CP and multiple CDNs. It also releases CPs from the technical difficulties of monitoring multiple CDNs and dynamically selecting the right ones to optimize their content delivery. It is expected that multi-CDN federation would attract more customers and bring a triple-win situation for the CDN vendor, the CP, and the end users.

Nevertheless, to form a full multi-CDN federation, CDN Interconnection (CDNI) [22, 23, 24] is required for dynamic traffic exchange among federated CDNs. CDNI requires a set of newly built interfaces and mechanisms to interconnect multiple CDNs such that the downstream CDNs (dCDN) is able to deliver content on behalf of the upstream CDN (uCDN). The interconnected CDNs not only need protocols and interfaces to exchange dynamic information (e.g., footprints and capabilities), but also need to replicate content from uCDN to dCDN. All these pose technical barriers. As a result, despite the active development in CDNI, *full multi-CDN federation has not become an industrial reality yet.*

The performance of content multihoming can be greatly improved, if information regarding traffic demand is available and resources from multiple CDNs can be leveraged. Research on the delivery of video streams has concluded that a centralized control plane could improve the performance of CDNs [2, 8], so we begin with a simple example to illustrate the initial motivation of our work. As shown in the top of Fig. 2.1, the system consists of two CDNs (CDN1 and CDN2), two CPs (CP1 and CP2), and two groups of end users ($e1$ and $e2$). Assume that i) each CDN has two PoPs and each CP can use both CDNs (so content multihoming is enabled), ii) $e1$ accesses content from CP1 and $e2$ accesses content from CP2, and iii) the traffic



Traffic distribution at time 3 (Content multihoming)

	a	b	c	d	Delivery latency
CP1	50	10	0	30	510
CP2	0	50	40	0	

Traffic distribution at time 3 (with centralized control)

	a	b	c	d	Delivery latency
CP1	50	40	0	0	390
CP2	0	20	40	30	

Figure 2.1: A motivating example: e_1 accesses content from CP1; e_2 accesses content from CP2; both CP1 and CP2 rely on CDN1 and CDN2 for content delivery.

demands to CP1 and CP2 over time are shown in the bottom left of Fig. 2.1.

Delivery with content multihoming: Each CP tries to get the maximum benefit of its own. In this case, most resources (service capabilities) of PoPs b and c will be consumed by CP2 before time 3, since b and c are the two nearest PoPs from $e2$. When the traffic demand of CP1 gradually increases from time 1 to time 3, the overall traffic demand of CP1 and CP2 reaches the peak at time 3. At this time, CP1 has to turn to PoP d to supply one third of its traffic demand, because no resources are available at other PoPs. The hop distance (i.e., the number of intermediate network devices that the traffic passes) between end user $e1$ and PoP d is relatively long, leading to a high accumulated content delivery latency[1] [33] for CP1. As a result, content multihoming is optimal only for CP2 but not globally optimal when both CPs are considered.

Delivery with centralized control: A centralized control with global view of the multi-CDN resources can make a globally optimal traffic and resource assignment. In this case, at time 3, a portion of CP2's traffic could be moved to PoP d such that the resources of PoP b can be shared with CP1. Therefore, the overall accumulated

[1]The accumulated content delivery latency is calculated by the traffic amount times its delivery hop distance.

content delivery latency of CP1 and CP2 can be reduced accordingly.

For simplicity, we only illustrate the snapshot on delivery latency at time 3 at the bottom right of Fig. 2.1. We can see that the accumulated content delivery latency with a centralized control is 390, much smaller than that with content multihoming, which is 510.

To improve the content delivery performance of the current multi-CDN architecture, we propose a *CDN semi-federation* solution where the federated CDNs are independent of each other. The CDN semi-federation model requires an authoritative and trusted third-party consortium formed by *voluntary* CDN vendors, which need to disclose their dynamic information (e.g., PoP footprints and service capabilities) to the consortium. The authoritative consortium adopts a centralized control to provide traffic delivery guidance to CDNs by leveraging the resources from multiple CDNs and reshaping the traffic demand assigned to each CDN. Compared with CDNI, CDN semi-federation: i) releases the CDN vendors from the complex technical and business obstacles of interconnecting with multiple dCDNs, and ii) avoids the sub-optimal content delivery decisions made by distributed CDNs, because CDNs will obey the centralized delivery guidance from the consortium instead.

2.2 Related Work

Related work on multi-CDN delivery can be roughly divided into two categories: content multihoming and CDN federation.

In the first category, CPs use multiple CDNs to deliver their content to maximize their benefit. The control decisions on traffic delivery are made either by the CP itself or by a CDN broker. In [6], Liu et al. used content multihoming to improve the performance of content delivery, while considering the price of different CDNs. Cost minimization for content multihoming has also been studied in [5]. They jointly considered the cost of using multiple CDNs together with the electricity cost in data centers and solved the integer linear optimization problem with an approximate algorithm.

Measurement-based studies on content multihoming are conducted in [9, 34], where the strategies in selecting multi-CDNs are analyzed based on measurement traces. For example, In [35], Gardner et al. analyzed a method of reducing latency by sending redundant requests to multiple CDN servers and using the server with the smallest response time. These studies mainly focus on improving the content delivery

performance to individual users rather than the performance of the overall network.

In the category of CDN federation, multiple CDNs form a federated union, and members in the federation pool their resources together. CDN federation extends the PoP footprint of single CDN and has stimulated many industrial activities. For instance, the Jet-Stream CDN project [36] intends to integrate multiple CDNs into its management software, so that through sharing contents and enlarging service scale, each member in the CDN federation can benefit from lower maintenance fee and higher profit.

Full CDN federation relies on CDNI, which has ongoing standardization efforts [22, 23, 24]. The research efforts of CDNI can be dated back to the early 2000s, when it was called Content Distribution Internetworking (CDI) [20]. In [19], Biliris et al. proposed the concept of CDN Brokering, which allows one CDN to dynamically redirect clients to other CDNs. Then the IETF working group on CDI [20] further defined the model of CDI [37]. This model was obsoleted by the framework of CDNI [22] in 2014.

Recently, there are reinvigorated efforts on CDNI sponsored by IETF [38], trying to define more technical details, e.g., framework, logging interface, and control interface/triggers. However, the optimization on content delivery performance in CDNI has not been well researched. CDN vendors make their own decisions on dCDN selection in CDNI architecture, which may lead to globally suboptimal decisions on content delivery. Besides, business relationships among CDNs might be highly complex due to the frequent role changes of a given CDN.

2.3 Architecture of CDN Semi-Federation

2.3.1 CDNI Background

To better understand the architecture of CDN semi-federation, we first summarize the main architecture of CDNI. The left side of Fig. 2.2 illustrates an example of content delivery with CDNI, where two interconnected CDNs (CDN-A and CDN-B) make a contract with CP1 and CP2, respectively.

Assume that an end user requests the content of CP-1. Since CDN-A is the contracted CDN of CP-1, the request is forwarded to CDN-A (uCDN) (Step 1 in the figure). Assume that this request is new and thus the requested content is not immediately available in either CDN's cache servers. The uCDN can decide to handle the request itself or use a dCDN to process the request, based on criteria such as

whether cache servers in uCDN are overloaded, or whether a PoP of the dCDN is closer to the user. Assume that the dCDN is closer to the user and thus the uCDN decides to use the dCDN (CDN-B) to deliver content on its behalf. In this case, CDN-A redirects the request to the dCDN (CDN-B) (Step 2). As the dCDN does not have the content in its cache servers, the content has to be firstly acquired from CP1 by CDN-A (Step 3), since only CDN-A has a contract relationship with CP1. The content then is transferred to the cache sever of dCDN (Step 4) and is finally delivered to the end user from the cache server of the dCDN (Step 5).

We can see that CDNI faces several key problems. First, the connected CDN vendors need real-time information exchange, e.g., the dynamic service capabilities and PoP footprint of each CDN. Thus, CDNI requires a set of interfaces (e.g., logging, footprint/capacity advertisement, and content acquisition) between interconnected CDNs. Second, uCDNs make independent decisions on selecting dCDNs, and a CDN can be either a uCDN or a dCDN, depending on the requested content. In the example shown in Fig. 2.2, if the content requested by the end user is from CP-2, then CDN-B becomes the uCDN. How to share profits between uCDN and dCDN is still under investigation, and the complexity is further compounded with the frequent role changes of a given CDN.

2.3.2 CDN Semi-federation

Now we introduce the framework of the proposed CDN semi-federation. As shown in the right side of Fig. 2.2, this architecture mainly consists of three components: 1) *traffic analyser*, 2) *central optimizer*, and 3) *consortium dispatcher*. Instead of asking each CP to make a contract with individual CDNs, we only need the CPs to have a contract with the consortium of CDN semi-federation.

The traffic analyser continuously estimates the traffic patterns of CPs from each end user location area, based on real traffic statistics. Traffic patterns of CP refer to the time-varying traffic demand for each CP, and will be an important input to the central optimizer.

As shown in Fig. 2.2, requests directed to the consortium will be collected by the traffic analyser. The traffic analyser classifies the requests according to: i) the location area where a request originates, ii) which CP the requested content belongs to, and iii) the time of request. This classification allows the traffic analyser to estimate the traffic patterns of each CP at different locations (by adding up the size of all

the requested contents in each class). Substantial research efforts have been devoted to traffic modeling and forecasting, and as such we assume that the traffic analyser relies on existing mechanisms (e.g., ARIMA [39]) to obtain the most up-to-date traffic patterns. A comprehensive study of traffic modeling and forecasting is beyond the scope of this chapter.

Based on the obtained traffic patterns, the *central optimizer* computes the content delivery schedule based on factors such as end user location, time of request, type of content, and so on. The optimization solution from the central optimizer will be sent to the *consortium dispatcher* for traffic dispatch.

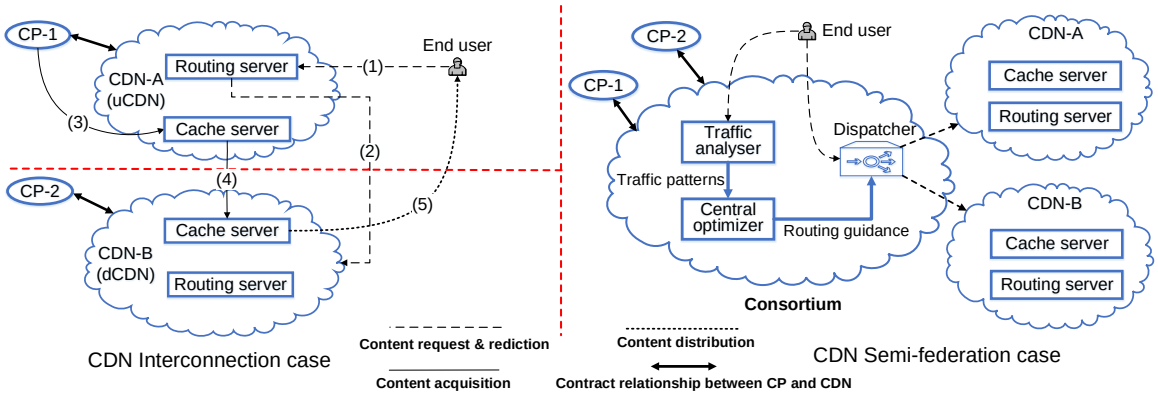


Figure 2.2: Difference between CDN Interconnection and CDN semi-federation

Existing request routing mechanisms (HTTP redirection, DNS `CNAME`) can be used by the *consortium dispatcher* for traffic dispatch. For example, if it uses the DNS-based redirection, DNS server of content provider needs to modify its `CNAME` record to point to the domain of consortium dispatcher instead of CDN-provided domain. The consortium dispatcher then uses another `CNAME` record to re-direct user requests to an optimal CDN PoP. Fig. 2.3 shows an example of video access to illustrate the procedure of CDN redirection.

The core component in the CDN semi-federation is the central optimizer, which needs to make globally optimal traffic delivery decisions based on factors such as users' demand patterns, the content types, and the status of PoPs. In practice, the computation in central optimizer can be performed over long intervals (e.g., several hours or one day). We disclose our solution to the design of the central optimizer in the next section.

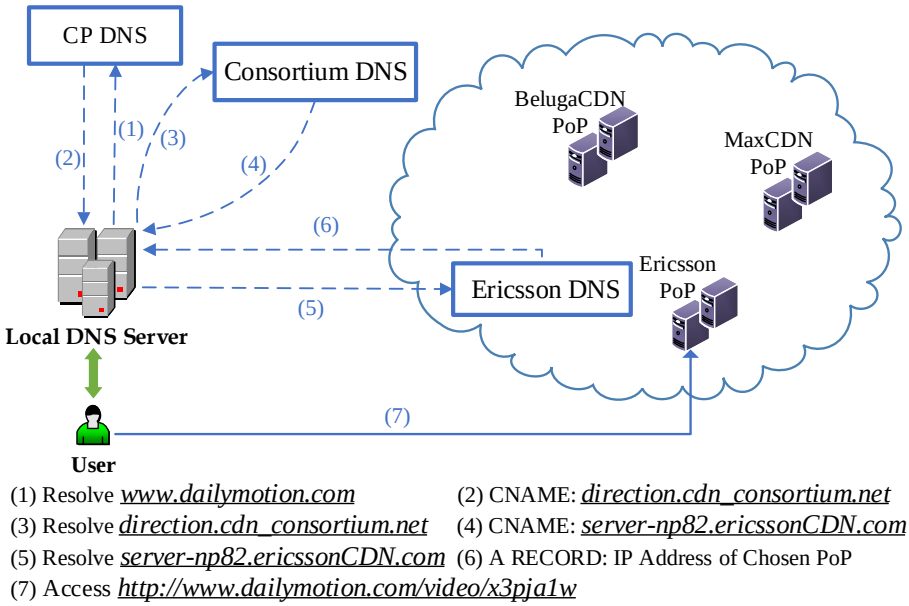


Figure 2.3: An example for DNS-based request routing. The central optimizer finds that the optimal solution for the user to access the CP's video is via the Ericsson PoP, so it notifies the consortium dispatcher to redirect the user requests to the Ericsson PoP.

2.4 Problem Formulation and Solution

To ease reference, the main notations used in this section are listed in Table 2.1, where the first part includes all the sets, the second part includes all variables, and the third part includes self-defined functions.

2.4.1 Composition of Target Network

We make use of ISP PoP network as the target network topology of our model. Since ISP PoP provides more detailed geographic information of traffic than AS-level network, the content delivery delay estimated by PoP hop distances can be more accurate than that estimated by AS hop distance [40]. Besides, each ISP PoP node is regarded as one location area of end users, who generate traffic demands to CDNs. Furthermore, since CDN vendors usually place their cache servers inside the ISP network, each ISP PoP node in our target network can also be a potential location of CDN PoPs.

Generally, there are three main players in the system: (1) end users, (2) content

Table 2.1: Main notations used in § 2.4

Term	*Definition*
T	A time period of n consecutive time slots
P	Set of all CPs in CDN semi-federation
V	Set of all CDN PoPs in CDN semi-federation
A	Set of location areas where end users are distributed
c_k	The maximum amount of traffic that PoP k can serve in a single time slot
h_a^k	Hop distance between PoP k and end user location area a
p_a^k	Performance of PoP k for demands from location area a
$\hat{p}_i$	Minimum performance required by CP i
$\alpha_{ai}^k(t)$	Dispatch guidance: fraction of traffic demand of CP i from end user location a assigned to PoP k at time t
$d_{ai}(t)$	The traffic demand of CP i from end user location area a at time t
$s(a,k,i)$	$s(a,k,i) = 1$ if PoP k can support the demands from location a with a performance higher than required by CP i ($p_a^k \geq \hat{p}_i$); $s(a,k,i) = 0$ otherwise
$r_k^i(t)$	Reshaped traffic demand from CP i assigned to CDN PoP k at time slot t
$\bar{r}_X(i)$	accumulated traffic demand from CP i served by CDN X during time period T
$\mu_i(t)$	Average delivery distance of content from CP i at time slot t
μ_i	Average delivery distance of content from CP i during T

providers, and (3) CDN vendors (or simply CDNs).

- *End users:* We assume that end users are grouped according to their locations (e.g., cities) distributed across the target network. The traffic demands from these locations need to be supplied by the CDNs. We define A as the set of all locations in the target network and use $a \in A$ to represent a specific location.

- *Content providers:* Content providers (CPs) are the customers of CDN semi-federation. They provide end users with content and rely on CDNs to deliver the content to end users. We use P to denote the set of all CPs that make contracts directly with the semi-federation and $i \in P$ to label a specific CP.

- *CDNs:* There are multiple CDNs in the CDN semi-federation. They need to share information such as PoP locations and cache capacities. We use V to

denote the set of all CDN PoPs within the semi-federation, and $k \in V$ to denote a CDN PoP.

2.4.2 Model of Traffic Demand Reshaping

CDN semi-federation reshapes the traffic demand of each CP, and assigns the reshaped demand to multiple CDNs based on the global view. We consider two significant factors: *time and location* in our model.

Real-world traces over CDNs suggest that traffic demands of different CPs have different patterns, and different traffic patterns can lead to significantly different content delivery latency [40]. Since traffic patterns are represented as time series data, we need to consider time in our model[2].

The geographic location of the end user requests is another important factor needed to consider, because hop distance between end user location and CDN PoP greatly impacts the content delivery latency. Therefore, the goal of our model is to find the optimal time-varying dispatching (reshaping) strategy for the traffic demand of each CP at each end-user location.

We assume that the traffic demand in our target network comes from the set of end user locations A, and denote the traffic demand of CP i from location a as d_{ai}. In addition, we consider the traffic within a periodic time window T (e.g., one day), which consists of n consecutive time slots (e.g., hours). Therefore, the traffic demand of CP i from location a in time slot t is denoted as $d_{ai}(t)$, for $\forall a \in A, \forall i \in P, \forall t \in T$.

We need to make schedules that assign the traffic demand $d_{ai}(t)$ to the set of CDN PoPs V within the semi-federation. To label a schedule, we use the notation $\alpha_{ai}^{k}(t)$, which denotes the fraction of traffic demand $d_{ai}(t)$ that is supplied by PoP k in time slot t. In addition, we pose the following constraints:

$$0 \leq \alpha_{ai}^{k}(t) \leq 1, \forall a \in A, \forall i \in P, \forall k \in V, \forall t \in T, \tag{2.1}$$

$$\sum_{k \in V} \alpha_{ai}^{k}(t) = 1, \forall a \in A, \forall i \in P, \forall t \in T, \tag{2.2}$$

to indicate that the traffic demands at each location should be completely satisfied.

Since each CDN PoP has a limited cache capacity, we define the cache capacity

[2]Taking into consideration of time difference, we use coordinated universal time (UTC) in this chapter. Therefore, two traffic patterns may exist a time shift due to different time zones they locate in.

of a CDN PoP as c_k, which denotes *the maximum traffic it can serve* in a single time slot. Then we have the following constraint:

$$\sum_{i \in P} \sum_{a \in A} \alpha_{ai}^{k}(t) d_{ai}(t) \leq c_k, \forall k \in V, t \in T, \tag{2.3}$$

meaning that in any time slot, the overall traffic assigned to a CDN PoP should not exceed its cache capacity.

2.4.3 Performance Guarantee

The performance of CDNs varies for the content demand from different end user locations, thus we should make sure that the required performance by CP (e.g., throughput or latency) can be guaranteed with the traffic assignment in CDN semi-federation. Let p_a^k denote the actual performance of PoP k in area a, and $\hat{p}_i$ the performance required by CP i. If a traffic request from location a is assigned to CDN PoP k, then $p_a^k \geq \hat{p}_i$ should be satisfied.

We define a function $s(a, k, i)$ to represent this relationship between the required performance and actual performance, i.e.,

$$s(a, k, i) := \begin{cases} 1 \text{ , if } p_a^k \geq \hat{p}_i, \\ 0 \text{ , otherwise.} \end{cases} \tag{2.4}$$

Since we should not assign traffic to a PoP which cannot meet the required performance, we have the following constraint:

$$\alpha_{ai}^{k}(t) = 0, \text{ if } s(a, k, i) = 0, \forall t \in T. \tag{2.5}$$

2.4.4 Minimizing Latency of Content Delivery

When content is delivered from CDN cache servers to end users, content delivery latency is proportional to the number of hops the content is delivered. Therefore, we use network hop distance to evaluate the latency. Based on this consideration, the accumulated content delivery latency across the network can be estimated by the amount of traffic times its delivery hop distance. If we use h_a^k to denote the hop distance between PoP k and user location a, then the overall content delivery latency

Φ during a certain time period T could be formulated as following:

$$\Phi := \sum_{t\in T}\sum_{i\in P}\sum_{a\in A}\sum_{k\in V} \alpha_{ai}^{k}(t) d_{ai}(t) h_{a}^{k}. \tag{2.6}$$

The optimal traffic delivery problem that the central optimizer needs to solve for CDN semi-federation can be formulated as the following minimization problem:

$$\min_{\{\alpha\}} \quad \Phi \tag{2.7a}$$

$$\text{s.t.} \quad (2.1), (2.2), (2.3), \ \textit{and}\ (2.5). \tag{2.7b}$$

Note that for the mathematical convenience in optimization, constraint (2.2) can be written as:

$$\sum_{k\in V} \alpha_{ai}^{k}(t) \geq 1, \forall a \in A, \forall i \in P, \forall t \in T. \tag{2.8}$$

One can also add additional constraints to (2.7) so as to provide CPs with more flexibility, e.g, the constraint to ensure that content of certain CP can be only accessed in specific geographical locations.

Remark 1. *We use the accumulated content delivery latency as the objective for the following reasons: i) the latency is critical for content delivery and a common concern of CDNs, CPs and end users, ii) reducing the content delivery latency aligns with the initial purpose of using CDNs, and iii) minimizing the overall content delivery latency is equivalent to minimize the transit cost charged by ISP in existing delivery distance/destination based billing models [41, 42].*

2.4.5 Problem Transformation

The large number of constraints formed by the four dimensional unknown variable $\alpha_{ai}^{k}(t)$ in preceding optimization problem (2.7), making it not readily solvable with existing solvers. We next refine the formulation of this problem, which derives a lower dimension optimization problem by eliminating the time factor t.

To ease understanding, we illustrate our solution with a simple scenario where only one CP and a single time slot (i.e., $|P| = 1, |T| = 1$) are considered. In this case, there is only one type of content accessed at each location area and the traffic demand from each area could be supplied by the set of CDN PoPs within target network A. Therefore, original $\alpha_{ai}^{k}(t)$ now becomes α_{a}^{k} with only two dimensions.

Assume that $|A| = x, |V| = y$, then the traffic supply guidance within our network can be expressed in the form of matrix as following:

$$\boldsymbol{\alpha} := \begin{bmatrix} \alpha_1^1 & \alpha_1^2 & \cdots & \alpha_1^y \\ \alpha_2^1 & \alpha_2^2 & \cdots & \alpha_2^y \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_x^1 & \alpha_x^2 & \cdots & \alpha_x^y \end{bmatrix}, \tag{2.9}$$

where each α_a^k means the fraction of traffic demand from user location a assigned to PoP k. For example, if the n^{th} line of matrix $\boldsymbol{\alpha}$ is equal to $(0.5, 0.2, 0.3)$, then CDN semi-federation will reshape the total traffic, and assign 50% of the traffic at location area n to CDN PoP 1, and the rest 20% and 30% will be assigned to PoP 2 and PoP 3, respectively (assuming the number of CDN PoPs is 3). Similarly, we use matrix $\boldsymbol{H}$ to represent hop distance between each end user location and each CDN PoP:

$$\boldsymbol{H} := [\boldsymbol{h}(1), \boldsymbol{h}(2), \cdots, \boldsymbol{h}(c)]^\intercal, \tag{2.10}$$

where each $\boldsymbol{h}(i)$ denotes a vector recording the hop distance from location area i to each of the CDN PoPs. Let vector $\boldsymbol{d}$ denote the amount of traffic demand from each of these end user locations. The optimization objective function Φ (accumulated content delivery latency) for this simple scenario can be formulated as following:

$$\Phi := \left\| \boldsymbol{d}^\intercal (\boldsymbol{H} \circ \boldsymbol{\alpha}) \right\|_1, \tag{2.11}$$

where the symbol "$\circ$" denotes the Hadamard product of two matrix, and each element of vector $\boldsymbol{d}^\intercal(\boldsymbol{H} \circ \boldsymbol{\alpha})$ represents the overall traffic assigned to a certain CDN PoP.

After making the solution to the above simple example clear, we next introduce the solution to the general case that includes multiple CPs and multiple time slots (i.e., $|P| = m, |T| = n$). The general case with multiple CPs and multiple time slots could be considered as a virtual target network A', which consists of $m * n$ A's. Each location set A in the virtual network represents an aforementioned simple network, in which only one specific content type is accessed for a specific time slot. The traffic assignment fraction variable $\alpha_{ai}^k(t)$ could be extended as follow:

$$\widehat{\boldsymbol{\alpha}} := [\boldsymbol{\alpha}(1), \boldsymbol{\alpha}(2), \cdots, \boldsymbol{\alpha}(m * n)]^\intercal, \tag{2.12}$$

in which each $\boldsymbol{\alpha}(j)$ represents the traffic assignment fraction matrix (refer to (2.9)) for a certain CP i in a specific time slot t, where $m*(t-1)+i=j$.

Similarly, we can extend the traffic demand vector for the case involving multiple CPs and multiple time slots, i.e.,

$$\widehat{\boldsymbol{d}} := [\boldsymbol{d}(1), \boldsymbol{d}(2), \cdots, \boldsymbol{d}(m*n)]^{\intercal}, \tag{2.13}$$

in which each $\boldsymbol{d}(j)$ represents a traffic demand vector $\boldsymbol{d}$ for CP i at time slot t, where $m*(t-1)+i=j$. Since the hop distance matrix $\boldsymbol{H}$ is fixed no matter which CP or time slot is considered, we simply expand the distance matrix $\boldsymbol{H}$ by $m*n$ times:

$$\widehat{\boldsymbol{H}} := [\underbrace{\boldsymbol{H}, \boldsymbol{H}, \cdots, \boldsymbol{H}}_{m*n}]^{\intercal}. \tag{2.14}$$

Hence, the accumulated content delivery latency in the general scenario with multiple CPs and multiple time slots can be calculated as follow:

$$\Phi := \left\| (\widehat{\boldsymbol{d}})^{\intercal} (\widehat{\boldsymbol{H}} \circ \widehat{\boldsymbol{\alpha}}) \right\|_1. \tag{2.15}$$

Following the same procedure, the constraints in problem (2.7b) can also be reformulated by using the above expanded matrix and vectors.

The new representation of problem (2.7) allows us to solve it efficiently with existing LP solvers such as CVX Gurobi solver [43].

2.5 Further Discussion: Business Considerations

2.5.1 Traffic Accounting among Multi-CDNs

From the business aspect, how to share the revenue among the federated CDNs might be the most significant problem that CDN semi-federation needs to consider. A simple idea on this problem might be that CDNs share revenue from a CP according to its delivery contribution to that CP. In other words, the CDN that delivered a higher volume of traffic will be given more revenue.

In CDN semi-federation model, we can simply calculate the overall amount of traffic assigned to a CDN by using the optimization result of (2.7). For a PoP k in the CDN semi-federation, the reshaped traffic demand from CP i at any time instance

t can be calculated as:

$$r_k^i(t) := \sum_{a \in A} \alpha_{ai}^k(t) d_{ai}(t), \forall t \in T. \tag{2.16}$$

Thus, the overall amount of traffic from CP i assigned to an arbitrary CDN X during the considered time period T can be measured by:

$$\bar{r}_X(i) := \sum_{t \in T} \sum_{k \in X} r_k^i(t). \tag{2.17}$$

This $\bar{r}_X(i)$ can be regarded as the overall traffic volume from CP i delivered by CDN X. It is first reshaped by consortium and then further assigned to CDN X. CDN vendors can also easily verify the authenticity of this value (given by the consortium) by checking the traces collected from its PoP nodes.

2.5.2 SLA on Content Delivery Latency

The average content delivery latency, as an indicator of performance, can be regarded as an important service-level agreement (SLA) between the content provider and the CDN vendor. With CDN semi-federation, for any CP i, the average content delivery hops at time instance t can be calculated as:

$$\mu_i(t) := \frac{\sum_{a \in A} \sum_{k \in V} \alpha_{ai}^k(t) d_{ai}(t) h_a^k}{\sum_{k \in V} r_k^i(t)}. \tag{2.18}$$

This is actually a weighted average delivery distance based on the volume of traffic delivered along each delivering path. Moreover, the average content delivery distance for CP i during the considered time window T across the whole network can be calculated as:

$$\mu_i := \sum_{t \in T} \left[\mu_i(t) \cdot \frac{\sum_{k \in V} r_k^i(t)}{\sum_{t \in T} \sum_{k \in V} r_k^i(t)} \right], \tag{2.19}$$

where each average delivery distance of time instance t is multiplied by the ratio of traffic demand within a single time slot to the total traffic demand during T. The metric u_i can be given by CDN semi-federation as an important SLA to CPs.

2.5.3 Why Should CDN Vendors Join CDN Semi-federation?

Benefits for Joining

By joining CDN semi-federation, individual CDN vendors can extend their PoPs and improve the quality of content delivery without extra deployment cost [44]. Meanwhile, CDN semi-federation helps cut down the transit costs that CDN vendors need to pay ISP for the content delivery [45].

According to [42], the pricing models under current Internet transit market are based on various factors, such as how far the traffic is traveling, whether the traffic is "on net" (i.e., to that ISP's customers), and the region of the delivery destination. Generally speaking, the transit cost for a CDN is closely related to the content delivery distance, because the CDN may need to pay additional transit cost, if the content is served from remote surrogates.

Overall, CDN vendors can benefit from CDN semi-federation on both performance and delivery cost.

Trustworthiness of Consortium

In the CDNI architecture, a uCDN trusts multiple dCDNs for content delivery through logging interfaces defined in [46]. Compared with CDNI where CDN vendors make the content delivery decisions by themselves, CDNs in semi-federation need to disclose their confidential information (e.g., footprints and cache capacities) to the consortium, and obey the decisions from the consortium for content delivery. The trustworthiness of the consortium, therefore, is necessary for CDN semi-federation.

Generally, the consortium should be a completely independent entity, and should not be solely controlled by any participating CDNs. As such, building trust to the consortium should not be harder than building trust among participating CDNs. In addition, since the content delivery mechanism of CDN semi-federation is known to participating CDNs, we believe that the trustworthiness mechanism is easy to build.

2.6 Performance Evaluation

In this section, we conduct *trace-driven* simulations with matlab and CVX to evaluate the performance of CDN semi-federation. We simulate real-world ISP PoP network by elaborately selecting real-world CDNs and CPs and their topology. We also extract

real-world traffic from different content providers.

2.6.1 Target Network & Traffic

Target Network Topology

To perform a comprehensive evaluation of CDN semi-federation, we build the target network based on ISP PoP networks. The PoP location and network topology data are collected from the *Internet Topology Zoo* [47], which includes detailed information of more than 250 ISPs all over the world. The target network of our experiments is constructed across North America and Europe, and contains the PoP topology information of 124 ISPs (AT&T, Bell Canada etc.) and 1057 ISP PoP nodes. The topology of constructed network is shown in Fig. 2.4.

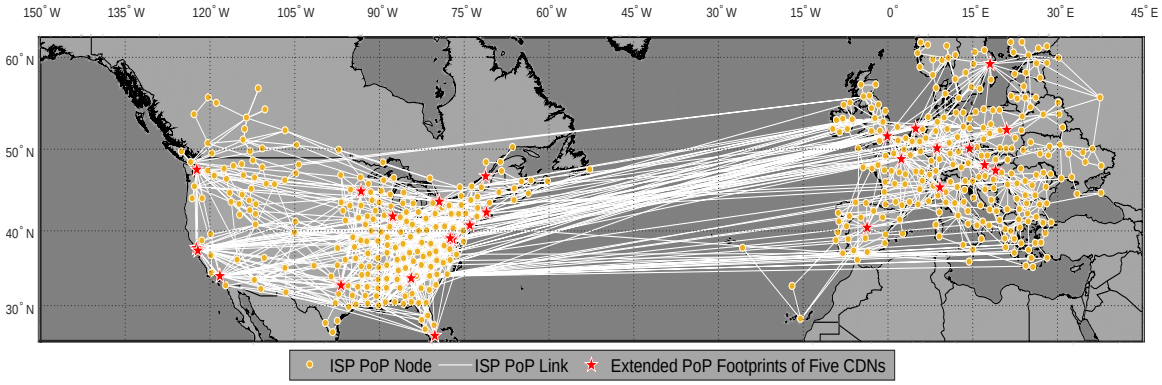


Figure 2.4: ISP PoP network across Europe and North America (Note that only half of the PoP nodes used in our experiments were drawn in order to make the figure clear).

According to the information from *CDN planet*[3], over twenty CDN vendors provide content delivery services in Europe and North America. Among them, we choose five popular CDN vendors, including MaxCDN, StackPath, CDN77, FastlyCDN, and BelugaCDN. The extended PoP footprints of the five CDNs are shown in Fig. 2.4. More detailed information of these CDN vendors can be found online from CDN planet.

Note that it is difficult to obtain accurate hop distances in the target network, because the routers between two nodes may change over time and some internal

[3]https://www.cdnplanet.com/

routers may not respond to ICMP queries. To overcome the problem, we use *weighted hops* to approximate hop distance by assuming that the hop distance between two nodes is proportional to their geo-distance [33, 48], i.e, for each hop we put a weight equal to the geo-distance (in km) of the hop. This approximation is based on the fact that round-trip-time (RTT) between two nodes is *approximately* linear with their geo-distance [48].

Traffic Demand of Content Providers

We consider five types of content: social media, e-commerce, gaming, online video, and crowdsourced live streaming. Accordingly, we select five representative CPs: Facebook, Amazon, Valve Software, Netflix, and Twitch, which mainly provide content in social media, e-commerce, gaming, video, and crowdsourced live streaming, respectively.

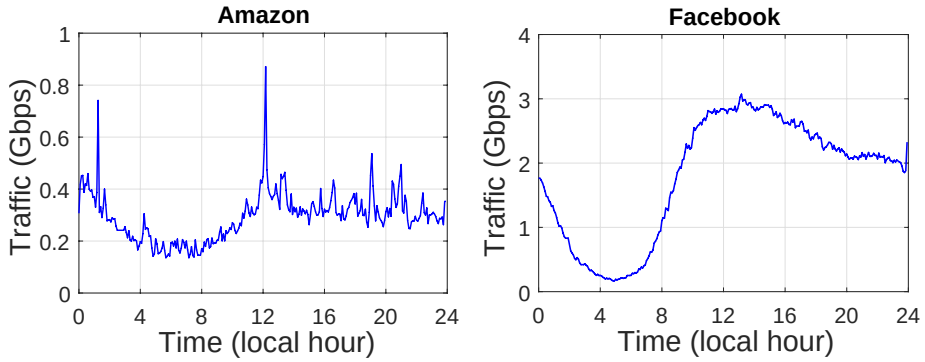


Figure 2.5: Traffic patterns of Amazon and Facebook on May. 07, 2017, extracted from NORDUnet.

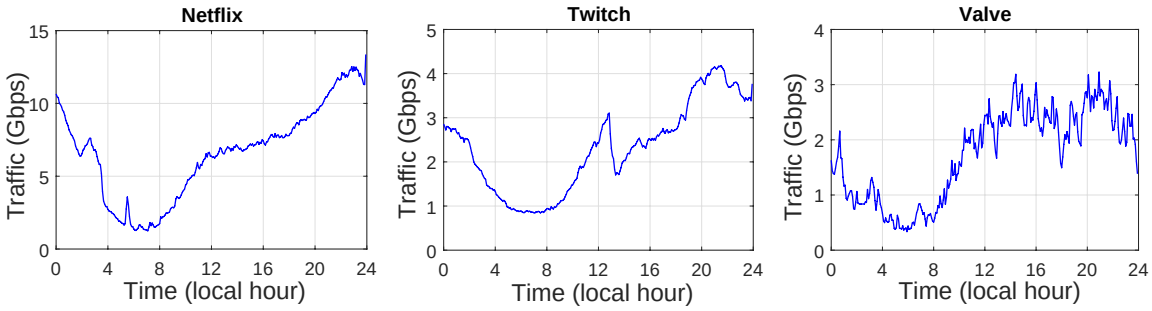


Figure 2.6: Traffic patterns of 3 CPs on May. 07, 2017, extracted from NORDUnet.

NORDUnet[4] is formed as a collaboration between the National Research and Education Networks of five Nordic countries. It hosts cache servers at various peering points over Europe and North America, and collects the traffic demand data of five aforementioned CPs. Fig. 2.5 and 2.6 shows the extracted traffic demand patterns (within a 24-hour periodic window) of the five aforementioned CPs. We treat these traffic demand patterns as the estimated traffic patterns obtained from the *Traffic Analyser* in Fig. 2.2 and feed them into the *Central Optimizer*.

2.6.2 Performance of CDN Semi-federation

We need to compare the performance of CDN semi-federation with that of other two main content delivery strategies: content multihoming and CDNI. With content multihoming, a CP uses multiple CDNs, and the CP determines the CDN that should be used for content delivery at a given time. To be more specific, every CP tries to maximize its benefits by selecting the CDN with the lowest latency. With CDNI, the performance is greatly determined by the dCDN selection strategy. As we mentioned before, although there are ongoing efforts on CDNI, the criteria for dCDN selection as well as the content delivery optimization in CDNI are not well researched. The current dCDN selection of CDNI is mainly based on the geographical distance to clients [49].

Therefore, current CDNI architecture shall obtain similar performance on delay as content multihoming, since both of them let a CP select a CDN PoP with the shortest hop distance to the clients. Thus, we only evaluate the performance of content multihoming in the rest of the chapter. Furthermore, we set a baseline for our evaluation, which is called *fixed contract* between CP and CDN. In other words, a CP will have contracts with a single CDN vendor to deliver its content without using multi-CDN strategy.

To test the performance of fixed contract, we assume several fixed bindings, in particular: Amazon with MaxCDN, Facebook with StackPath, Netflix with CDN77, Twitch with Fastly, and Valve Software with BelugaCDN. Note that based on this contract relationship, we set the total amount of allocated cache for a given CDN equal to the peak traffic demand of its contracted CP. To be more specific, for a given CDN, if its contracted CP is i, then the total amount of allocated cache for this CDN

[4]http://stats.nordu.net/connections.html

is calculated as:

$$\sum_{k \in V} c_k = max\{\sum_{a \in A} d_{ai}(t), \forall t \in T\}. \tag{2.20}$$

For the purpose of comparison, cache allocation method in (2.20) will be also adopted in the evaluation experiments of content multihoming and CDN semi-federation.

To test the performance of content multihoming, we need to consider the competition for CDN resources among different CPs. For this purpose, we develop an algorithm to calculate the content delivery latency with content multihoming. The algorithm adopts a greedy approach that a CP always selects the nearest PoPs (to the end users) for its content delivery. To be more specific, in every time slot the algorithm randomly picks a CP i and end user area a, and sets this combination (CP and area) the highest priority for accessing the CDN PoP resources. It then directs the traffic of CP i at area a to the nearest available PoPs from a (cache capacities of the PoPs decrease accordingly). Note that this traffic demand may be supplied by multiple PoPs due to the limited cache capacity of a single PoP. The algorithm repeats the above steps until there is no more new combination of CP and location area or all the PoPs are fully loaded.

We ran the experiment 100 times and observed that the obtained result shows a very small variance (more than 95% results locate in the range of $\pm 5\%$ around the mean value). Thus, we in the rest mainly show the average result of content multihoming taken over the 100 runs.

Overall Performance Comparison

The hourly traffic delivery latency is shown in Fig. 2.7, where we observe that content multihoming and CDN semi-federation have different performance. The difference becomes evident during two time periods. The first period occurs around UTC time 02:00-07:00, which is the night peak in North America (corresponding to 10:00 PM-03:00 AM for east coast, 07:00-12:00 PM for west coast). The second period is around UTC time 20:00-24:00, which is the night peak of western & central Europe. Fig. 2.7 indicates that when the system experiences high traffic demands, CDN semi-federation significantly outperforms content multihoming. This phenomenon is easy to understand: When the overall traffic demand is low, the CDN resources are abundant, and almost every end user can get the service from his nearest PoP, regardless content delivery strategies. Nevertheless, when the traffic demand is high, the above condition does not hold, and many users need to fetch content from remote PoPs. In

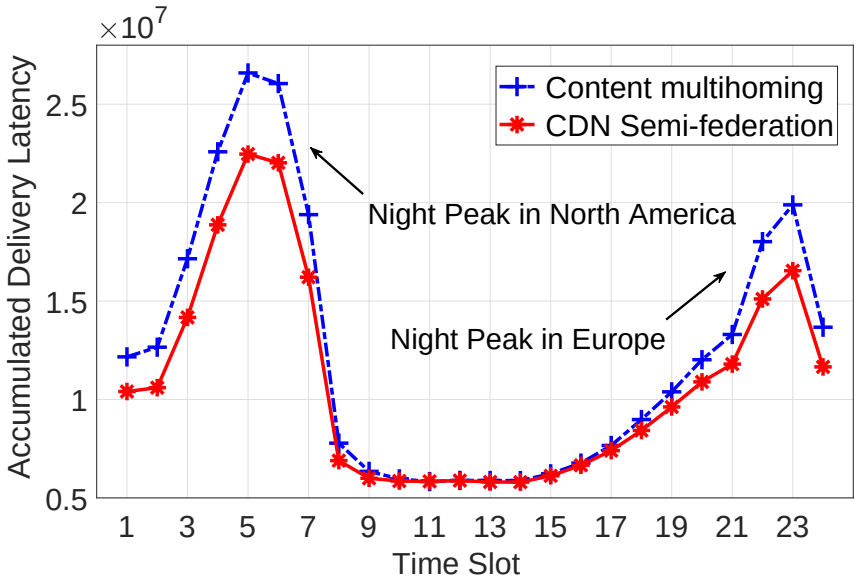


Figure 2.7: Hourly accumulated traffic delivery latency (Gb·weighted hops), where time slot 1 corresponds to UTC time 00:00 to 01:00 AM.

this case, CDN semi-federation has more global knowledge than content multihoming to make much better content delivery decisions.

The overall traffic delivery latency is shown in Fig. 2.8, in which we add the accumulated traffic delivery latency of different time slots together. We can see that compared with content multihoming and fixed contract, CDN semi-federation reduces the overall traffic delivery latency by 14% and 45%, respectively. As an importance indicator of content delivery latency, the average content delivery hop distances in our experiments are shown in Fig. 2.9, where CDN semi-federation outperforms content multihoming on content delivery latency.

Performance Comparison Over Peak Traffic Hours

We now narrow down to the two peak time periods shown in Fig. 2.7. We select 6 time slots with the highest overall traffic demand from end users, and compare the performance between content multihoming and CDN semi-federation. As shown in Fig. 2.10, CDN semi-federation reduces the average traffic delivery latency by 20% over content multihoming averaged during the time periods with peak traffic demands. Note that in our experiments, the traffic demands of CPs are designed can be 100% satisfied by the CDN resources during the peak traffic hours. In practice,

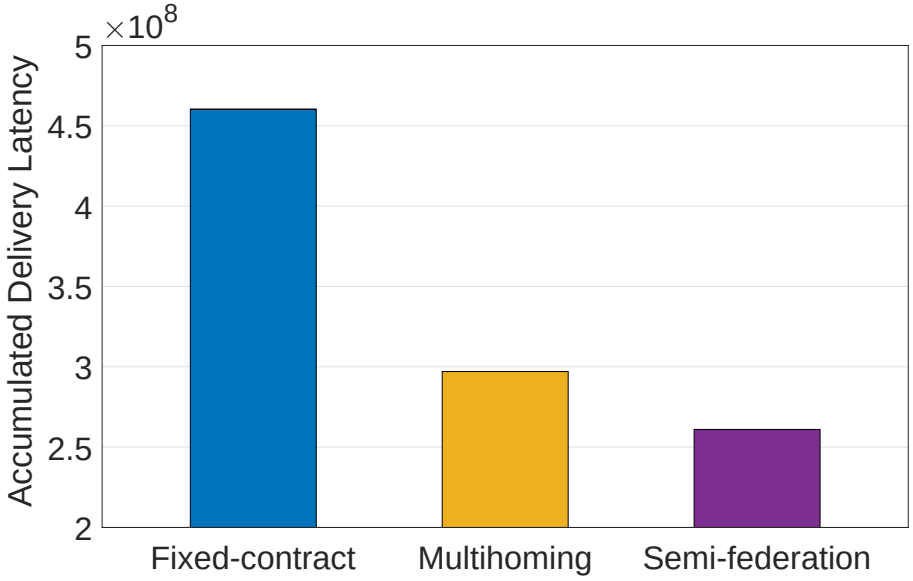


Figure 2.8: Overall traffic delivery latency (Gb·weighted hops) in one day.

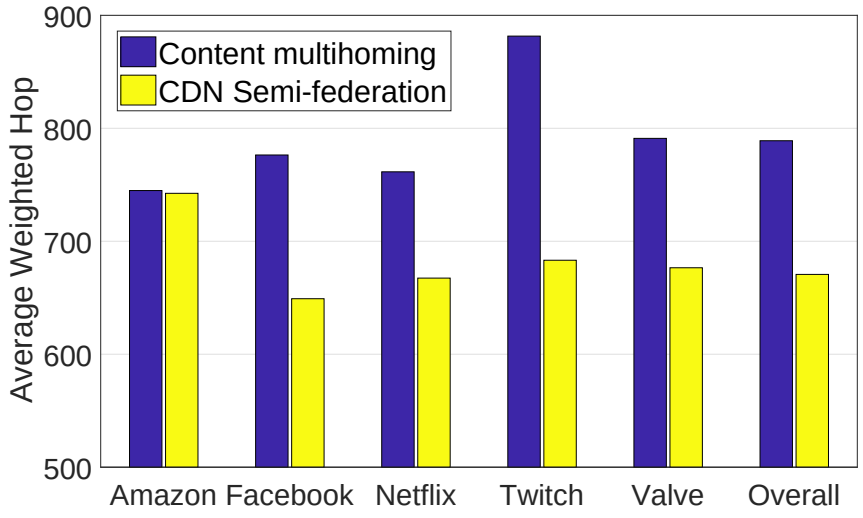


Figure 2.9: Average content delivery distance (weighted hops).

CDN semi-federation could be more efficient when the CDN resources are insufficient (i.e., unable to supply all traffic demands).

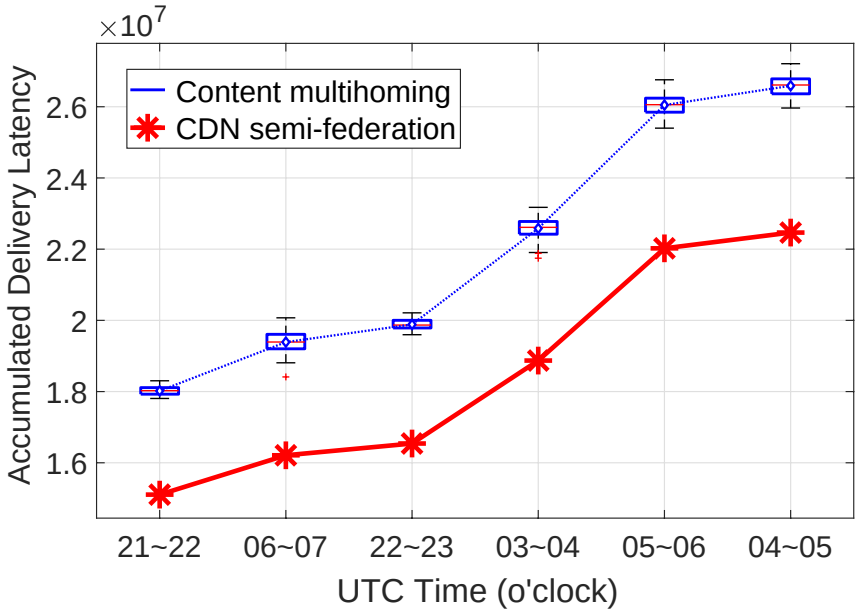


Figure 2.10: Latency during peak traffic hours.

2.6.3 Traffic Demand Reshaping

CDN semi-federation introduces many-to-many CP-CDN combinations that take advantage of diverse traffic patterns and broader PoP footprints for better content delivery performance. From the perspective of a single CDN, traffic pattern from each CP has been reshaped by CDN semi-federation to gain more benefits.

To illustrate traffic demand reshaping, we extract the data from each of the CDNs under our experiments. Fig. 2.11 illustrates the result of i) reshaped traffic patterns in MaxCDN under CDN semi-federation and ii) the traffic patterns in MaxCDN under content multihoming. Since content multihoming lacks global knowledge on different CDN's cache capacities, the CDNs with low cache capacities usually get fully loaded. We can see from Fig. 2.11 that even during the time period with small traffic demand, the traffic supplied by MaxCDN is still at a high level under content multihoming strategy, since MaxCDN owns the least cache capacities among all the CDNs.

Compared with content multihoming, CDN semi-federation shows a better performance on load balance by traffic demand reshaping. For a specific time slot, we define the *cache utilization* of a given CDN as the overall amount of traffic supplied by this CDN divided by its cache capacity c_k. Fig. 2.12 shows the variance of cache utilization of five CDNs in every two hours. We can see that the load of five CDNs

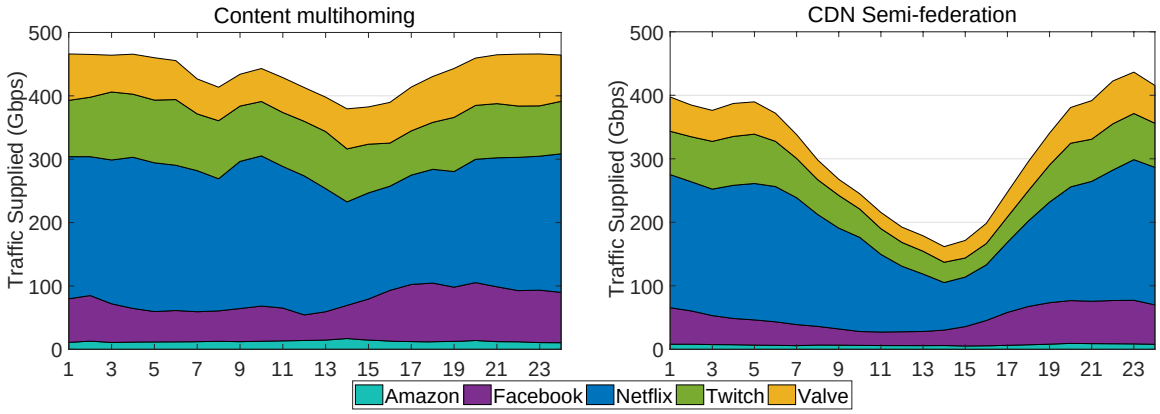


Figure 2.11: Reshaped traffic of five CPs supplied by *MaxCDN* during different time slots.

is more balanced under CDN semi-federation. Note that the average CDN cache utilization in 24 hours of CDN semi-federation and content multihoming is 71% and 78%, respectively.

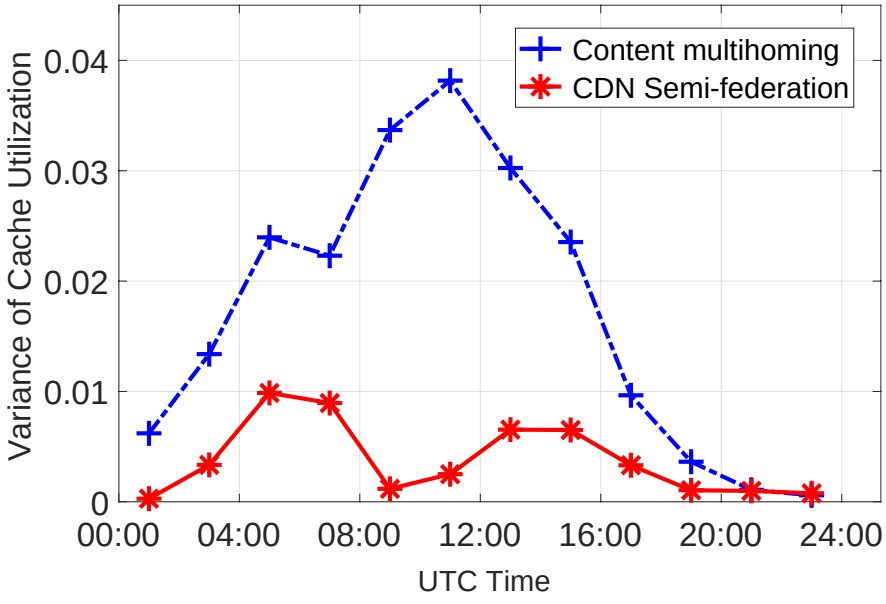


Figure 2.12: Variance of five CDNs' average cache utilization every two hours based on UTC time.

Comparing Fig. 2.7 and Fig. 2.11, we observe a very interesting phenomenon: the

overall traffic supplied by *MaxCDN* slightly decreases from time slot one to five (so are the other CDNs), whereas the traffic delivery (shown in Fig. 2.7) increases during this time period. This is due to temporary mismatch between traffic demands and local PoP capacity. Time slots one to five (equal to UTC time 00:00-05:00 AM) correspond to 02:00-07:00 AM in central Europe and 07:00-12:00 PM in central America. During this time period, the traffic demands of most CPs decrease in Europe, but increase over central and eastern U.S., which host most ISP PoP nodes of North America according to Fig. 2.4. The traffic demands in central and eastern U.S. are beyond the capacity of PoPs in this region and thus turn to the PoPs of western U.S. or Europe, leading to a long delivery distance. Even though the problem of temporary mismatch between demand and local PoP capacity is inevitable due to distributed demands, the results in Fig. 2.8 suggest that CDN semi-federation can alleviate this problem by traffic demand reshaping.

2.6.4 Strength of Combined CDNs

Benefit of Extending CDN PoP Footprint

We answer a key question: *with the same amount of overall cache capacity*, can extending footprint of CDN PoPs help lower the average content delivery latency? The question is important to justify the benefit of CDN semi-federation, as CDN semi-federation implies the extension of individual CDN's PoP footprints while their total cache capacity remains the same.

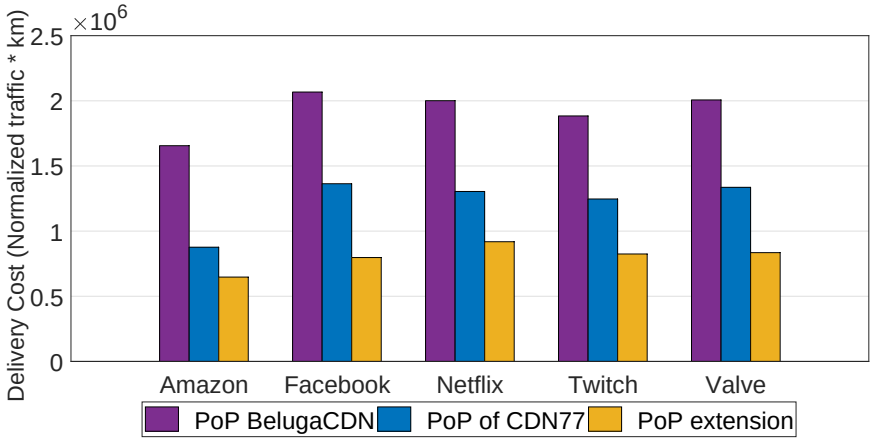


Figure 2.13: Overall traffic delivery latency. Note that the total amount of allocated cache in the *PoP extension*, *CDN77*, and *BelugaCDN* is equal.

To answer this question, we specifically choose two CDNs: *CDN77* and *BelugaCDN*. *BelugaCDN* has good coverage in North America (11 PoPs) while a weak coverage in Europe (3 PoPs), whereas *CDN77* has more balanced PoP locations (9 and 6 PoPs in Europe and North America, respectively). Furthermore, we assume there is a virtual CDN owning all the PoP locations of *CDN77* and *BelugaCDN* (labeled as *PoP extension* in Fig. 2.13). Since the traffic volume of different CPs varies greatly, e.g., the average traffic volume of Netflix is nearly 20 times higher than that of Amazon, to easily compare the average content delivery latency of different CPs, we normalize a CP's hourly traffic amount with its one-day's total volume, making the sum of hourly traffic within one day equal to one. The performance is shown in Fig. 2.13. From the results, we find that increasing the footprint of PoPs indeed leads to improvement on content delivery performance.

Combining Advantages of Different CDNs

We further test the impact of traffic patterns on content delivery performance. For this test, we distribute the traffic of each CP (normalized) over different CDNs. A metric, *relative* delivery latency, is defined to show the comparative results of different CPs over a particular CDN. Given a CDN and a set of CPs using this CDN, the *relative* delivery latency for a specific CP i is calculated as:

$$(latency(i) - avg\_latency) \,/\, avg\_latency,$$

where $latency(i)$ denotes the accumulated content delivery latency of CP i over the given CDN. The average delivery latency $avg\_latency$ equals $\frac{\sum_{i \in P} latency(i)}{|P|}$, where $|P|$ is the number of CPs using the given CDN.

The result is shown in Fig. 2.14. We can observe that CDNs have their own advantages in delivering different types of content. For example, over CDN77, Facebook has the highest latency among all the CPs, while Netflix becomes the most time-consuming one when delivered over StackPath. This phenomenon further confirms the strength of CDN semi-federation: advantages of different CDNs should be combined for maximum benefit.

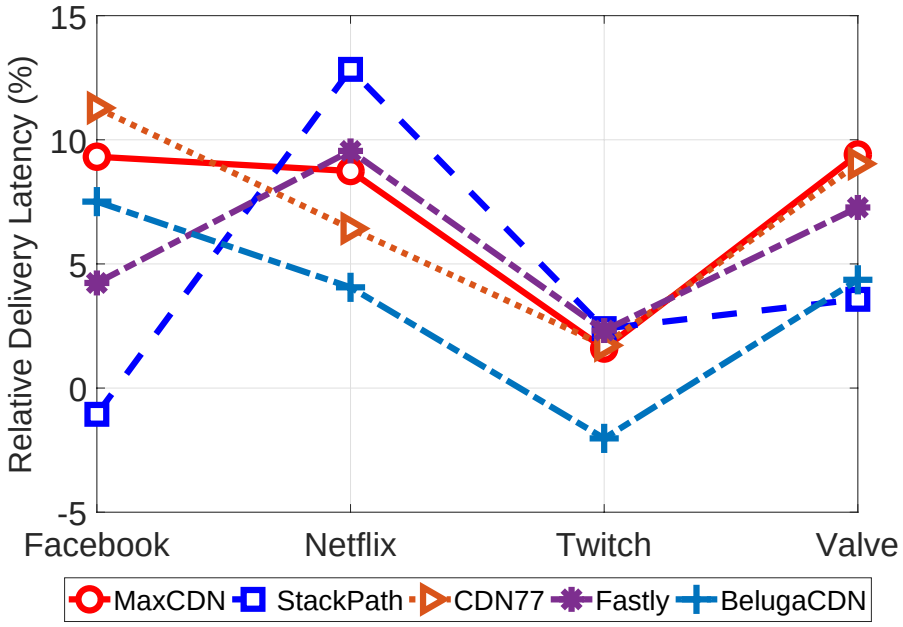


Figure 2.14: Relative delivery latency of different content types by different CDNs, where the traffic patterns of different CPs are normalized.

2.7 Conclusions

In this chapter, we introduced the CDN semi-federation framework as an alternative implementation of CDNI to speed up the multi-CDN content delivery. We tackled the core difficulty in CDN semi-federation, the optimal traffic dispatch, by considering the combined PoP footprints, the distance between end user locations and PoPs, and different traffic demand patterns. Live video providers could rely on the CDN semi-federation to further improve the QoE of their live viewers with our multi-CDN delivery strategy. In the following two chapters, we will propose two more specific solutions to improve the QoE of live videos delivered over the edge CDN servers.

Chapter 3

Edge-Assisted QoE Optimization of HTTP-based Live Video Delivery

In this chapter, we propose a detailed scheme to optimize the QoE of HTTP-based live videos. Our scheme uses a real-time exploration and exploitation (E2) model to learn the most suitable initial video segment for the viewers.

3.1 Introduction

Live large-scale events drive nearly 10 times more viewer engagement than on-demand videos, and the HTTP-based live streaming has been gaining increasing popularity in recent years [1]. Due to the stringent real-time requirement and high traffic spikes of live videos, content delivery networks (CDNs) continue to struggle with delivering high-quality live videos to viewers while guaranteeing their Quality-of-Experiences (QoE) [2, 3]. Since live viewers are normally very sensitive to QoE deterioration such as slow startup time and the dreaded spinning pinwheel (i.e., video buffering), lower QoE generally means a higher abandonment rate of users.

To address such an issue, CDN operators rely on the widely distributed edge servers (e.g., the edge data centers [4]) to handle the increasing demand of live videos. Using edge servers as the cache, most viewers can fetch the requested contents directly from the cache rather than the original video sources. Nevertheless, the first batch of requests would still miss the edge cache due to the real-time property of live streaming [11]. Even worse, the performance of TCP-based content transmission may decline due to the long-latency backhaul in the content delivery path [50].

The existing research mainly uses adaptive bitrate algorithms to improve the QoE of viewers via dynamically choosing a bitrate for each video segment [51, 52, 53]. While these efforts have shown considerable QoE improvement, they may lead to decreased video quality during the streaming process. Many studies have pointed out that users may quickly abandon a video session if the quality is not sufficient [54, 55]. An alternative method to improve the QoE is to conduct transient holding of a minimum number of video segments at the edge servers such that the clients can select a suitable initial video segment (*IVS*) that best matches their network throughput [18] to start the playback. Our experience in real-world live video streaming suggests that this is a much more effective way for QoE improvement.

Existing *IVS* selection strategies either use a fixed value [28] or use the "optimal" value [18]. In the former, the RFC standard of HTTP Live Streaming (HLS) suggests that "client should not choose a segment that starts within *three* segment durations (the maximum playback duration of video segments in the playlist) from the end of the playlist file" in order to avoid playback stalls [28]. In the latter, the "optimal" *IVS* value is derived to match the current network conditions (e.g., throughput) [18].

Clearly, the former will not work well for high-quality live video streaming due to the dynamic network conditions. The latter is promising but has two main pitfalls. First, it relies on the *per-user* based network throughput estimation. The network throughput is related to multiple complex factors (e.g., RTT and router buffer size), which frequently changes over time [29]. This can incur high computational overhead, especially for live videos where the number of live viewers is large [11]. Second, when a user joins a live channel, the server can only infer the user's network throughput through the signal strength (e.g., RSRP, RSRQ and RSSI in LTE) and mobility pattern (e.g., fast, slow, static). The network throughput estimation in this case may not be accurate, leading to a suboptimal choice of *IVS*. In practice, the overhead of searching for the "optimal" value may offset the benefit. It is also hard to quickly react to the network condition change.

To overcome the above problem, we propose a reinforcement learning-based dynamic *IVS* selection scheme (*Rldish*) deployed on edge CDN server to maintain a balance between exploring suboptimal decisions and exploiting currently optimal decisions. *Rldish* uses a real-time *exploration and exploitation (E2) model [30] to learn the IVS selection automatically, and is deployed as a virtualized network function (VNF) on the CDN edge server by the CDN operator. It can work seamlessly with existing edge CDN proxy (cache) server (e.g., Nginx) [11, 31], and can also react to the network*

condition (throughput) change via real-time exploration.

Rldish makes the *IVS* decisions on a *per-stream basis* to avoid high overhead in per-user based throughput estimation. Since an edge CDN server generally serves its proximal end users, viewers accessing the same live video usually share the common video delivery path from the origin server to the edge and generally experience the similar network conditions when fetching the same video from the edge. Based on this observation, *Rldish* continuously updates the currently optimal decisions on *IVS* selection for the live viewers on a *per-stream basis*, based on the real-time QoE measurements and feedback. The decisions will then be updated into the media playlist files of each stream for the subsequent live viewers.

Our goal is to implement *Rldish* only at the edge servers and make no changes on either the client or the origin server. In the following sections, we gradually tackled the following non-trivial research challenges.

First, how to mitigate the negative impact of caching on the learning performance? It is uncertain that whether an exploration action will lead to a cache hit at the edge server. In other words, the same *IVS* may miss the edge cache sometimes but hits the cache at other times. Since cache hit or miss would greatly impact the QoE of viewers, it may "confuse" RL due to significant different rewards fed by the same choice.

Second, how to conduct real-time QoE measurements at the edge? RL requires feedback of viewers' QoE to compute the reward of previous explorations. However, the viewers' QoE metrics (e.g., startup latency, video buffering ratio/time) are generally collected at the client side. Modifying the client side to collaborate with the edge is not desirable.

Third, how to support different types of QoE objectives? Live videos consist of different types of traffic (e.g., live event streaming, and user-generated live videos). Viewers may have different QoE metrics for different contents. For example, live event streaming might value the lower playback latency more.

3.2 Related Work

Related work on QoE optimization of live videos can be primarily grouped into two categories: QoE enhancement via bitrate adaption and network throughput prediction, and optimization of *IVS* selections.

In the first category, research efforts have been devoted to adaptive live stream-

ing algorithms. In [56], Bruneau-Queyreix et al. proposed a prototype of a hybrid P2P/multi-server video quality-adaptive streaming solution, which simultaneously uses multiple servers and peers to enhance the QoE of live viewers with expanded bandwidth and link diversity. Aiming at satisfying live viewers' personalized QoE, Wang et al. [57] proposed a reinforcement learning-based solution to automatically learn the transcoding selections so that the backhaul bandwidth could be saved. To reduce the general latency of live videos, the authors in [58] proposed an adaptation bitrate algorithm for HTTP-based live streaming by conducting TCP throughput prediction. The research of QoE optimization of live videos has also been conducted in [59, 51] and [60].

In the second category, Ge et al. [18] improved the QoE of live viewers by holding a minimum number of video segments at the edge server such that the clients can select the optimal *IVS* to best match their network throughput. While this work has shown decent QoE improvement, it may have two pitfalls. First, it relies on the *per-user* based network throughput estimation to derive the optimal *IVS* values, which may lead to high computational overhead. Second, this scheme may not quickly react to network throughput changes.

3.3 Background and Overview of Rldish

3.3.1 HTTP-based Live Video Delivery

HTTP-based live video streaming has gained increasing popularity in recent years owing to that HTTP is compatible with large numbers of client-side applications (e.g., web browsers and mobile applications) [3]. As illustrated in Fig. 3.1, once a raw live video is generated from the source (i.e., the broadcaster), it is first uploaded to the origin server, where it is encoded into multiple streams with different pre-determined bitrates. The server then splits each stream into a sequence of small video segments. To watch videos, the clients download the segments sequentially with HTTP GET [28, 2].

Every time when a client joins a live channel, she first needs to request the playlist file (shown in Fig. 3.1) from the origin server which contains a list of up-to-date (i.e., the segments that can be readily fetched) video segments of the requested live channel. Based on this playlist, the client then chooses an initial video segment to start the playback [28]. Afterwards, the client generally plays the video segments in the order

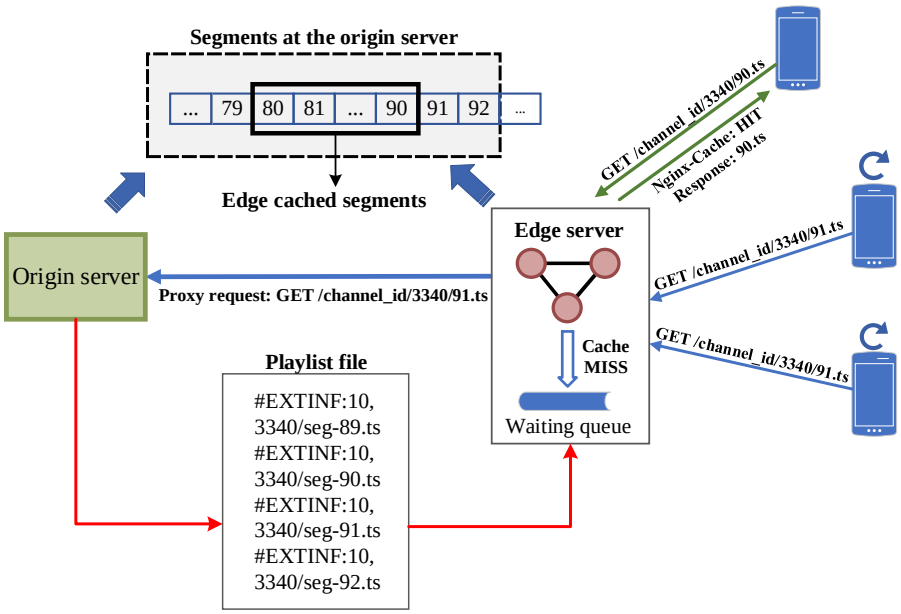


Figure 3.1: Architecture of live video delivery over edge servers.

that they appear in the playlist.

For live streaming, this playlist file is continuously updated by the streaming server once new segments have been generated. The client continuously accesses the newest playlist file to know the up-to-date video segment information (i.e., URI). Since caching video segments at the edge server is generally triggered by viewer requests and the video segments need time to be delivered from the streaming server to the edge server, it is possible that the newest several video segments may not be fully cached in the edge server when their information is already shown in the playlist file (e.g., segment 91^{th} and 92^{th} in Fig. 3.1).

3.3.2 The Impact of *IVS* on QoE of Live Viewers

The selection of *IVS* can impact the QoE of live viewers greatly. We use the example in Fig. 3.1 to explain. The figure shows a scenario where segments of a live stream (with bitrate 3340 kbps) from sequence number 0^{th} to 92^{th} have all been generated at the origin server, while the newest two segments 91^{th} and 92^{th} have not yet been cached at the edge server. At the same time, three new live viewers near the edge server try to join this live channel based on the information of the fetched playlist.

Although playing the newest segment (segment 92^{th}) would provide the users

with the smallest streaming latency, it may generate *playback stalls* for the watching process later on, since there are no pre-buffered segments at the edge that can be used for the subsequent segment requests of the client. Since the edge server typically issues a new HTTP request to the origin server to fetch the segment once a cache miss happens, this also induces a high *startup latency* for live viewers. A naïve idea to reduce the high startup latency and remove the buffering events is to join the live channel with a relatively conservative *IVS* (e.g., segment 89^{th} in Fig. 3.1). Nevertheless, a too conservative *IVS* may lead to unnecessary latency (i.e., a receiver's playback is far behind the live streaming source).

Therefore, it is critical to develop a dynamic *IVS* selection policy to optimize QoE. The policy needs to consider both the complex caching status and real-time network conditions. Note that QoE of live viewers is mainly determined by backhaul throughput under the cloud-edge streaming infrastructure because viewers generally have sufficient download speeds from their local edge CDN servers. One might consider that when the backhaul throughput is under a poor condition, QoE of live viewers will get worse again after a certain period of video watching even if it started with the right *IVS*. Actually, the proxy (edge) server normally maintains multiple keep-alive TCP connections with the streaming server. Thus, the download processes of multiple video segments will be conducted simultaneously if the backhaul throughput is less than the average bitrates of live stream. Therefore, the edge server can ensure the client, once it selects the right *IVS*, to have local access to the subsequent video segments [18].

3.3.3 Overview of Rldish

Rldish realizes the dynamic *IVS* selection policy within the edge server. We present the core design and the implementation of *Rldish* in Fig. 3.2. It consists of three key components:

1. *QoE Collector*: it communicates with the (HTTP) proxy server in real time to collect the QoE data (e.g., startup latency, video buffering time) of live viewers.

2. *RL based IVS Selector*: it accepts the fresh data of the rewards feedback from the *QoE Collector* to continuously update the latest *IVS* selection for each live stream by running a reinforcement learning algorithm.

3. *Playlist Manager*: it periodically sends requests to the origin server for the

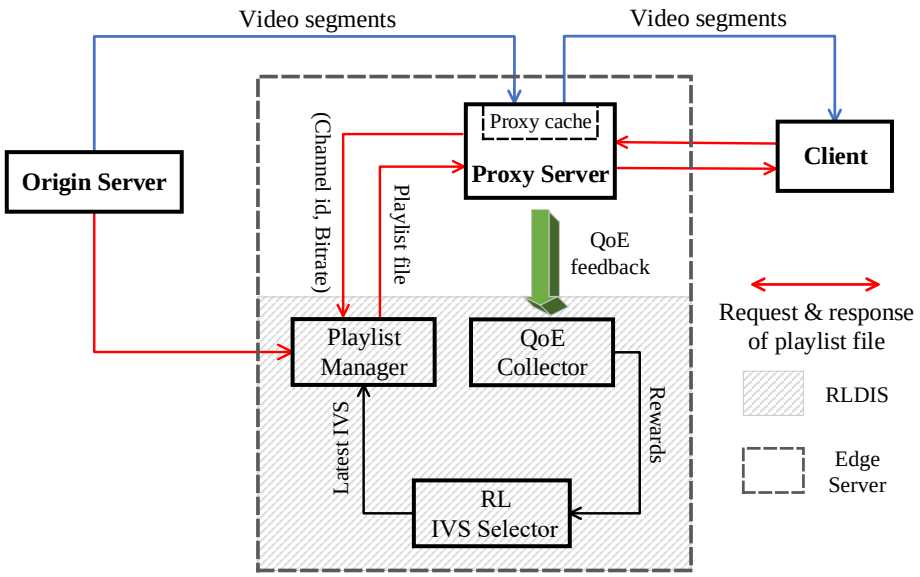


Figure 3.2: System design of *Rldish*.

most up-to-date playlist files of live streams currently accessed by local users. It maintains the playlist files of all live streams in the local cache by using the new *IVS* (from the RL) to update the original playlist files.

When a user joins a live channel, the proxy server will locate the first request for the user's playlist file to the local edge cache. By identifying the requested channel ID and bitrate information from the request URI, the corresponding playlist file for the requested stream can be obtained from the *Playlist Manager*. Once the first requested playlist file has been successfully delivered to the client, the subsequent requests for playlist files are handled by the proxy server independently: it returns the up-to-date unmodified playlist file to the client. In practice, there are many ways for the HTTP proxy server to distinguish whether a playlist request is from a new client or not so as to take different response decisions. One of the easiest way is to use an HTTP Cookie to indicate the state information of a connection.

After the user has played the live stream for a certain time (e.g., 3 minutes after the first playlist request), *QoE Collector* collects data (e.g., startup latency, total buffering duration) from the proxy server by analyzing the transmission finish time of each individual segment and their expected playback time at the client end. The collected QoE data, as the feedback of the previous trial, are reported to the RL module, which provides the *Playlist Manager* with the newest *IVS* information to

update the playlist files for new coming live viewers. In this way, *Rldish* could control the playback behaviour of all viewers served at the edge. *Note that clients can still adjust their video bitrates during the watching process to obtain better QoE*, since *Rldish* only manipulates the *IVS* selection step.

In the following, we introduce the core RL algorithms used by the *IVS Selector*.

3.4 Core Algorithms

3.4.1 Discounted-UCB (D-UCB) for Non-stationary MAB

The problem of dynamically learning and choosing *IVS* in real time according to time-varying network conditions can be modelled as the non-stationary multi-armed bandit (MAB) problem [61], where the bandit facing K arms needs to decide which arm to play when the environment may change (i.e., the distribution of rewards may change) over time. This model is consistent with our situation where the network conditions may change over time (for both paths from the origin to the edge and from the edge to the end users). We modify Discounted-UCB (D-UCB) [61], a variant of UCB (Upper Confidence Bound) algorithm, to solve this problem. For the content to be self-contained, we introduce the details of D-UCB.

The D-UCB algorithm can adapt to the QoE drift of the live streaming, since it automatically gives higher weight to more recent measurements by exponentially discounting historical measurements with a discount factor [30, 61]. The details of D-UCB are shown in Algorithm 1, where $T \to \infty$. At each time t, the bandit chooses an arm $I_t \in \{1, \cdots, K\}$ with the highest instantaneous expected reward $\bar{X}_t(\gamma, i)+c_t(\gamma, i)$, where $\bar{X}_t(\gamma, i)$ (shown in Equation (3.1a)) is the discounted empirical average of the observed rewards, and $\gamma \in (0, 1)$ is a discount factor to control the algorithm's preference degree for the recent measurements. The smaller the γ, the higher discount rate on the historical measurements, therefore the higher weight on recent measurements. $X_s(i)$ denotes the instantaneous reward of arm i at time s.

$$\bar{X}_t(\gamma, i) = \frac{1}{N_t(\gamma, i)} \sum_{s=1}^{t} \gamma^{t-s} X_s(i) \mathbb{1}_{\{I_s=i\}} \tag{3.1a}$$

$$N_t(\gamma, i) = \sum_{s=1}^{t} \gamma^{t-s} \mathbb{1}_{\{I_s=i\}}, \tag{3.1b}$$

$$\mathbb{1}_{\{I_s=i\}} = \begin{cases} 1 \text{ , if } I_s = i, \\ 0 \text{ , otherwise.} \end{cases} \tag{3.1c}$$

$c_t(\gamma, i)$ is the discounted padding function, which is defined as follows:

$$c_t(\gamma, i) = 2B\sqrt{\frac{\xi \log n_t(\gamma)}{N_t(\gamma, i)}}, \; n_t(\gamma) = \sum_{i=1}^{K} N_t(\gamma, i), \tag{3.2}$$

where B is the upper bound on the rewards and $\xi > 0$ is a parameter to control the probability of exploration. When an arm is frequently used in the past, its padding function gets smaller than the other arms, so that the other arms get a chance of being explored [29].

Algorithm 1: Discounted UCB

```
for t from 1 to K, play arm I_t = t;
for t from K + 1 to T do
    play arm I_t = arg max_{1≤i≤K} X̄_t(γ, i) + c_t(γ, i).
```

3.4.2 Tailored D-UCB Algorithm

The D-UCB shown in Algorithm 1 needs to keep all the historical rewards (i.e., $X_s(i), \forall s \in \{1, \cdots, t\}$, $i \in \{1, \cdots, K\}$) to calculate $\bar{X}_t(\gamma, i)$ and $c_t(\gamma, i)$ for each time step, which may result in high computational overhead. Considering the limited resource of edge server and high traffic volume of live streaming, we need to tailor the D-UCB algorithm to solve our problem more efficiently.

Let $\hat{X}_t(\gamma, i) := \sum_{s=1}^{t} \gamma^{t-s} X_s(i) \mathbb{1}_{I_s=i}$. We have $\bar{X}_t(\gamma, i) = \frac{\hat{X}_t(\gamma,i)}{N_t(\gamma,i)}$. Instead of recalculating $\hat{X}_t(\gamma, i)$ and $N_t(\gamma, i)$ with the historical rewards each time, the calculations could be completed easily using their previous states, i.e.,

$$N_t(\gamma, i) = \gamma^{\Delta_i(t)} \times N_r(\gamma, i) + \mathbb{1}_{\{I_t=i\}}, \tag{3.3a}$$

$$\hat{X}_t(\gamma, i) = \gamma^{\Delta_i(t)} \times \hat{X}_r(\gamma, i) + X_t(i)\mathbb{1}_{\{I_t=i\}}, \tag{3.3b}$$

where r is the last time (before t) that arm i was selected and $\Delta_i(t) = t - r$. For a given γ and time t, we represent the $N_t(\gamma, i)$ and $\hat{X}_t(\gamma, i)$ of all arms in the format of vector, i.e., $\boldsymbol{N} = [N_t(\gamma, 1), \cdots, N_t(\gamma, K)]$ and $\boldsymbol{X} = [\hat{X}_t(\gamma, 1), \cdots, \hat{X}_t(\gamma, K)]$, to allow easy updates on $\bar{X}_t(\gamma, i)$ and $c_t(\gamma, i)$. Then the instantaneous expected rewards for all the arms at time t can be computed as:

$$\mathbf{R} := \boldsymbol{X} \oslash \boldsymbol{N} + 2B((\xi \log \|\boldsymbol{N}\|_1)\boldsymbol{N}^{\circ -1})^{\circ \frac{1}{2}} \tag{3.4}$$

where $\oslash$ and $\circ$ denote the entrywise division and power operation, respectively. The details of the tailored D-UCB is shown in Algorithm 2, where $\boldsymbol{e}_i$ denotes $(0, \cdots, 0, 1, 0, \cdots, 0)$ where the i^{th} element is 1 and all other elements are 0.

Algorithm 2: Tailored D-UCB

```
X = N = R = 0 ∈ R^K
for t from 1 to T do
    play arm I_t = t (If t ≤ K)
    play arm I_t = arg max_{1≤i≤K} R (Otherwise)
    X_t(i) ← Get the reward of I_t, i ← I_t
    X = γX + X_t(i)e_i, N = γN + e_i
    calculate R (refer to Equation (3.4))
```

3.4.3 Definition of Reward Function

In order to use RL to dynamically select the best *IVS* to optimize QoE, we need to define the metrics that measure the QoE of live viewers. While QoE may be measured in many different angles and there is no consensus on the QoE metrics, we use the most-adopted ones for QoE evaluation in live streaming [62, 63], including:

- *General latency (gl)*: the delay time that the viewer's playback is behind the video source's live production progress.

- *Startup latency (sl)*: the delay between the time when the user sends the first request and the time when the playback starts.
- *Buffering time (bt)*: the total time of playback stalls experienced by the live viewer.

The above three metrics together are sufficient to describe the QoE of live viewers. In addition, these metrics are closely related to the *IVS* selection: A high startup latency is normally caused by the cache miss of the first requested segment (*IVS*) in the edge server; a high general latency is mainly caused by a too conservative *IVS* (i.e., segment far away from the end of original playlist); a long buffering duration is mainly due to a too aggressive *IVS* (i.e., segment too close to the end of original playlist). Combining all the above three metrics, our reward function is given in Equation (3.5), where $\alpha + \beta + \delta = 1$ and $0 \leq X_t(i) \leq 1$.

$$X_t(i) = 1 - (\alpha * \frac{sl_t(i)}{sl_{max}} + \beta * \frac{gl_t(i)}{gl_{max}} + \delta * \frac{bt_t(i)}{bt_{max}}). \tag{3.5}$$

In reward function (3.5), $sl_t(i)$, $gl_t(i)$, and $bt_t(i)$ denote the startup latency, general latency, and buffering time of arm i at time t, respectively. Accordingly, sl_{max}, gl_{max} and bt_{max} denotes the maximum startup latency, the maximum general latency, and the maximum buffering time observed in the history, respectively. α, β and δ are the weight factors of the three different QoE metrics. In practice, since different live video providers may emphasize different QoE metrics (e.g., streaming of live events may favor lower general latency), they could customize this reward function by using different values of scale factors (i.e., α, β and δ).

3.4.4 Definition of Arms

The D-UCB algorithm requires a decision arm space with discrete values, which is in accord with our problem with discrete video segment choices. An intuitive idea is to define an arm as the gap between a certain segment and the last segment (in the current playlist). However, this method would induce a problem over the edge-cloud delivery infrastructure. Since for a given *IVS* using arm x (segment with x segment distance to the playlist end), it is uncertain that whether the subsequent user requests to this segment will hit the edge cache or not. When a request hit the edge cache, it generally provide RL with decent rewards feedback. While when the cache miss

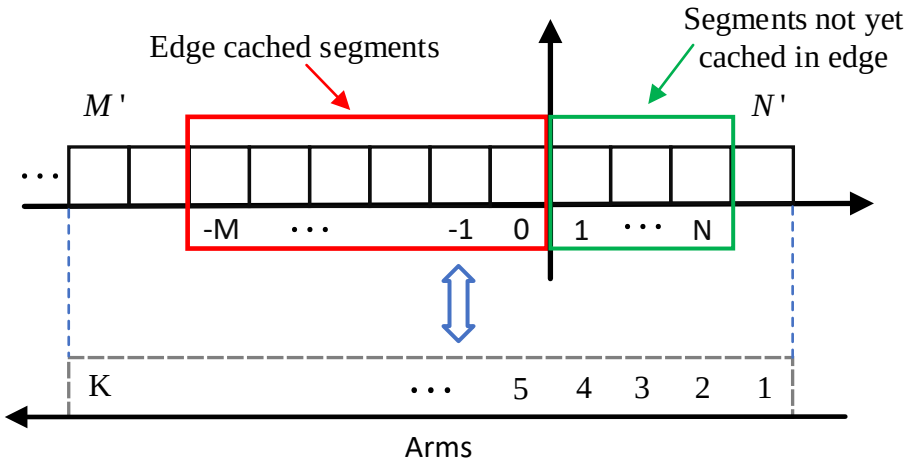


Figure 3.3: Illustration of arm definition for RL.

happens at the edge, QoE rewards from the same arm could get much worse. The very different feedback on the same arm could "confuse" the RL and prevent it from providing good decisions.

To solve this problem, we define the arms by *considering the real-time position of video segments in the edge cache instead of their position in the playlist.* Fig. 3.3 shows the general caching status of a live stream at an arbitrary time instant, where each video segment is denoted by a square (the closer to the right, the newer the segment). Video segments that have already been cached in the edge are marked by the red box, and the segments that have been generated (shown in the playlist) but have not been cached in the edge server are marked by the green box. To ease illustration, we label the segments from $-M$ to N, where $-M$ and N correspond to the oldest segment cached in the edge and the newest segment shown in the playlist file, respectively, and 0 refers to the newest segment currently in the edge cache. Note that since edge caching is a dynamic process triggered by user requests, M and N are dynamic (non-negative) values during the whole streaming process.

We then select a fixed, sufficiently large segment interval $[M', N']$ such that $[-M, N] \in [-M', N']$ holds all the time. Based on the new interval, we could setup a *one-to-one* mapping between the arm set $\{1, \cdots, K\}$ and $\{-M', \cdots, N'\}$, as shown in Fig. 3.3, where $K = M' + N' + 1$. Therefore, whenever an *IVS* choice ($I_t \in \{1, \cdots, K\}$) is given by RL, *Playlist Manager* first uses the above mapping to find the corresponding video segment in the playlist, then modifies the playlist file accordingly.

Table 3.1: HTTP Request & Response Data

Metrics	Description
Request Processing Time (rpt)	Time elapsed from when the edge receives the first byte of request until the last byte of response is acknowledged by the client
Upstream Response Time (urt)	Time elapsed from when the edge establishes a connection to an upstream server until it receives the last byte of the response body
Segment Size (ss)	Size of the requested segment
Round Trip Time (rtt)	Smoothed round-trip time (srtt) between the edge and the client
Response Finish Time (rft)	Local time when the last byte of response is acknowledged by the client
Caching Status	Segment request HIT or MISS at the edge cache

3.5 Implementation of Key Components

3.5.1 QoE Collector

To measure the QoE metrics from the edge side, *QoE Collector* collects the performance data of HTTP interactions of each live session from Nginx, as shown in Table 3.1, all of which can be tracked during the interaction process for each HTTP request. It then calculates QoE metrics, the startup latency, buffering time, and general latency, using the collected information.

Startup Latency (sl)

Fig. 3.4 shows the basic HTTP interactions between the edge and the client. Let l and s denote the request for playlist file and *IVS*, respectively. Based on the Caching Status of each HTTP requests (shown in Table 3.1), *Rldish* measures the startup latency of each live video session as follows.

When an *IVS* request misses the edge cache,

$$sl := \frac{t_4 - t_3}{ss} * \bar{ss} + (t_3 - t_1), \tag{3.6}$$

where ss and $\bar{ss}$ are the size of *IVS* and the average video segment size of this live

stream, respectively, and t_1, t_3, t_4 are calculated with the performance data collected (refer to Table 3.1 and Fig. 3.4) by $t_1 = rft_l - rtt - rpt_l$, $t_3 = t_2 + urt_s (t_2 = rft_s - rtt - rpt_s)$, and $t_4 = rft_s - rtt$, respectively. Note that $(t_4 - t_3)$ is the *IVS* transmission time, and $(t_3 - t_1)$ is the sum of the other startup time excluding the *IVS* transmission time.

It is worth mentioning that even for the same live stream (with the same bitrate), the sizes of generated video segments (with the same playback duration) are not exactly the same. Different sizes of *IVS* generally bring different segment transmission time, which implies different startup latency and different rewards. Our reward function implicitly considers the impact of different *IVS* sizes on RL by normalizing the video transmission time in (3.6). This is based on the fact that persistent HTTP connection between the edge and the origin server is generally adopted for live streaming, and thus the transmission time is (approximately) proportional to the segment size.

When an *IVS* request hits the edge cache, urt_s becomes (nearly) 0 and the startup latency could be calculated by $\frac{(t_4 - t_2)}{ss} * \bar{ss} + (t_2 - t_1)$ instead. Note that we use t_1 as the request starting time to calculate startup latency without including the time for (local) TCP connection establishment between the edge and the client. This is because: i) the connection establishment time is irrelevant to the arm selection, and ii) the time is much shorter than the *IVS* transmission time.

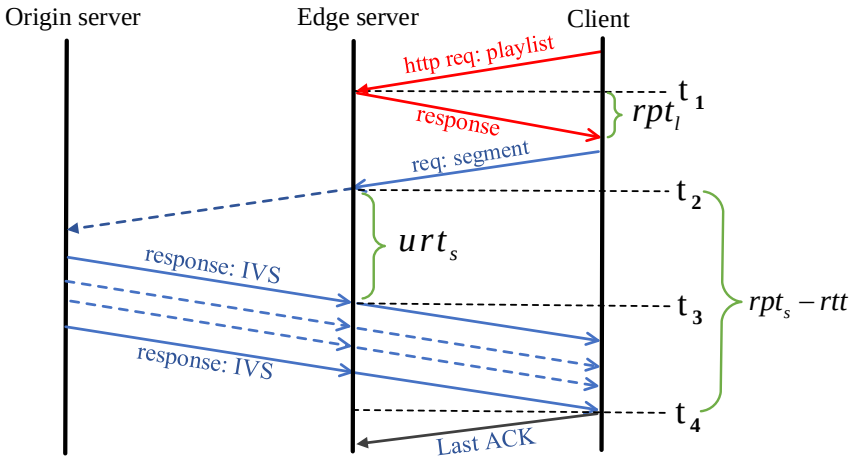


Figure 3.4: Illustration of HTTP interactions of live streaming between the client and the edge server.

Buffering Time (bt)

To calculate the buffering time of a live viewer, we collect the *rft* of all video segment requests during a certain time period from the beginning of the live session. The total buffering time could be derived as follows:

$$bt = \sum_{i=2}^{N} max\{rft_i - rpt_i, 0\}, \tag{3.7}$$

where rft_i is the response finish time of i^{th} video segment and rpt_i denotes the playback start time of i^{th} segment (the time when the playback of $(i-1)^{th}$ segment is finished). If we use pt_{i-1} to denote the real playback start time of $(i-1)^{th}$ segment, then we have $rpt_i = pt_{i-1} + sd$, where sd is the segment (playback) duration. We can calculate pt_{i-1}, recursively, with $pt_{i-1} = max\{rpt_{i-1}, rft_{i-1}\}$. Note that when $i = 2$, pt_{i-1} is the startup latency sl, and the corresponding rpt_{i-1} is 0 (no startup latency). In this recursive way, we can obtain the total buffering time.

General Latency (gl)

The general latency of a live session is defined as the segment duration time multiplied by the number of segments between *IVS* and the newest segment in the playlist when a user sends her first request. This metric can be easily calculated since all the related values are readily available. Note that in a given live stream, while the segments may have different sizes, their duration time is all the same (e.g., each segment lasts for 5 seconds).

3.5.2 Playlist Manager

The *Playlist Manager* generates the final playlist files that will be accessed by clients, and it needs to communicate with multiple units to accomplish that function. To reduce the overhead, the playlist file of a live stream is not generated individually for each new request. Instead, there are only *three types of events* in our system that can trigger the update of playlist file of a live stream: i) RL module provides a new choice (different from previous one) of *IVS* of the live stream, ii) a fresh new playlist file with the description of new segments is obtained from the origin server, and iii) new segments of the live stream are cached in the edge server.

The first two types of events are controlled by *Rldish* itself, while the last type of

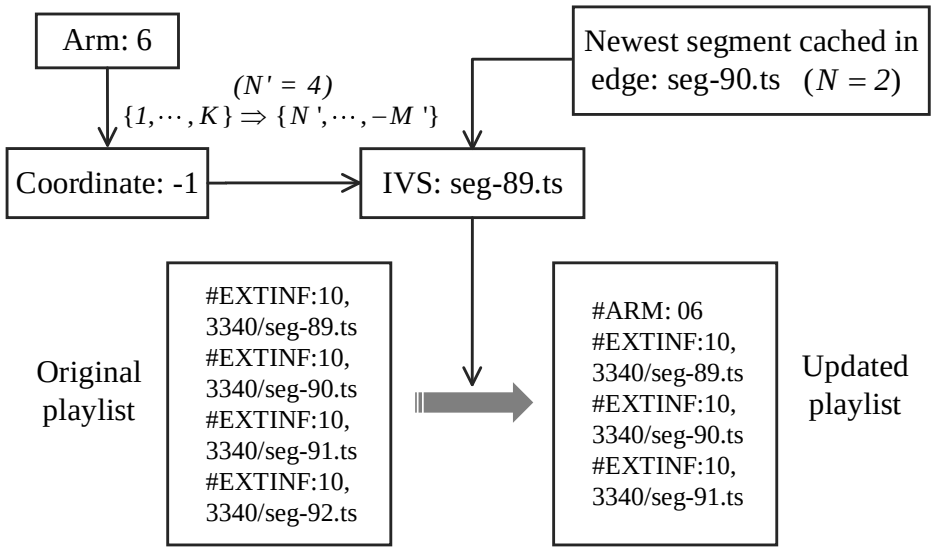


Figure 3.5: Example of playlist file update procedure.

events (i.e., edge caching) is managed by the Nginx server. To know exactly the edge caching status, the *Playlist Manager* monitors the edge cache in real time to find if new cache files have been generated and which video segments are in the cache files.

Using HLS protocol as the example, Fig. 3.5 shows an example on how to update the playlist file, where arm 6 given by RL will first be directed to video segment 89^{th}. Then we remove the information of segment 92^{th} from the original playlist so that the HLS client will play segment 89^{th} as the *IVS* to start the live streaming. Note that the information of 90^{th} and 91^{th} could also be eliminated from the updated playlist in this example. Also note that the HLS client by default starts the playback from fragment $N-2$, N being the last segment of the live playlist [28]. Thus, segment 90^{th} would be played as the *IVS* if the playlist is not updated by the *Playlist Manager*

We add a new tag `#ARM` in the updated playlist to mark the arm selection of each playlist file during the whole streaming process. The *QoE Collector* module will then obtain the value of this tag from the HTTP response to know the arm selection of each live session (persistent HTTP connection used), i.e., the mapping between arm selections and their corresponding QoE rewards, as introduced in § 3.4.3 and § 3.4.4.

3.6 Performance Evaluation

3.6.1 Experimental Setup

Clients & servers setup: We use HLS as the streaming protocol in our experiment. Based on the well-known hls.js (version 0.12.4) library [64], we implement the HTTP live streaming client relying on HTML5 video. Our client adopts the default HLS configuration which proactively buffers the future video segments when the duration of total buffered segments is less than 30 seconds or the total buffered segment size is less than 60 MB. The client runs on a Google Chrome browser (version 75) on top of Windows PCs.

The edge cache server is deployed on a local data center in Hong Kong with an average 8 ms RTT to our client machines. The bandwidth for a single segment download is set to 64 Mbps, which is in accord with general download speed from a local CDN edge server. It runs Nginx (version 1.9.9) as the HTTP reverse proxy. *Rldish* is implemented and deployed as a Docker [65] container on this edge server. Using the RL algorithm introduced in § 3.4, it accesses the QoE observations from Nginx and provides a revised playlist file to the Nginx through shared volume mounted from the host machine.

Since in practice the origin server managed by the content providers could be in different locations, we create 3 VM instances, located at East Coast North America, West Coast North America, and Japan, respectively, to serve as the origin streaming servers. The first two instances are launched on East Cloud and Arbutus Cloud of Compute Canada, respectively, and the third VM in Tokyo is launched via Google Cloud. The average RTTs from our edge server to the 3 VMs are 234 ms, 156 ms, and 52 ms, respectively. Each VM runs the system of Ubuntu 18.04 with kernel 4.15.0-34 as well as an Apache HTTP web server.

Live video setup: To generate the source of live video streaming, we use FFmpeg [66] to convert a local 2K High Frame Rate (HFR) video file (MP4 format) into 3 HLS live source with average bitrates 24 Mbps (2K HFR), 16 Mbps (2K standard), and 8 Mbps (1080P), respectively. Each live source is further generated into two HLS streams with a video segment length of 5 seconds and 10 seconds, respectively. We deploy the video segments of the 6 HLS streams (i.e., stream with quality 2K HFR, 2K standard and 1080p with segment length 5s and 10s respectively) onto each of the above-mentioned 3 origin servers. Then for each stream, we create a background pro-

cess (in each of the VM), which periodically (every 5 or 10 seconds) generates a new HLS playlist file according to the normal segment playback sequence (i.e., updating the playlist file by adding new segment information into it). The streaming process at the origin server is repeated, once it reaches the last segment of a video.

Test scenarios: To evaluate the performance of *Rldish* and other schemes, we set up multiple HLS clients. Each client joins a live stream by first sending the HTTP request to HTTP proxy server in the edge (in practice, DNS CNAME redirection is commonly used to accomplish this procedure). It then plays the live video for 2 minutes. The QoE data of the live viewers are collected during this period.

To evaluate the performance of *Rldish* under changing network conditions, we manually set up the streaming servers to limit the segment download throughput for a single request. When conducting our experiment on a given live stream, we setup the backhaul network throughput range from 1/3 the stream bitrate to the stream bitrate. Refer to Fig. 3.7 for more details on the throughput setting. For a given throughput, each of the clients repeatedly joins the live streaming and plays the video 30 times based on the real-time *IVS* decisions made by *Rldish*. The network throughput then changes to another value within the range, to evaluate if *Rldish* could quickly react to the network condition change. In order to collect sufficient QoE observations to evaluate the performance of *Rldish*, our HLS clients join each of the live streams broadcast by each of our streaming servers (18 live streams in total).

3.6.2 Evaluation Methodology

We evaluate the performance of *Rldish* and compare it with that of two other schemes:

ETHLE: this scheme was proposed in [18], which calculates the *IVS* choice (x) based on the following inequality:

$$\arg\min_{x} x \geq \frac{1}{l_{seg}}(d_{startup} + \frac{s_{seg} - th_{startup}}{bw_{max}}), \tag{3.8}$$

where l_{seg} and s_{seg} are the duration and size of a single video segment, respectively, $d_{startup}$ and $th_{startup}$ are the duration and total number of bytes transferred in the slow start period of TCP, respectively. bw_{max} is the bottleneck bandwidth of the backhaul link. We assume this scheme has the complete network throughput information (i.e., our backhual throughput setting) without using the estimation method. Refer to [18] for more details of the algorithm.

E2E: this scheme serves the live viewers via the CDN edge server directly. It does not adopt dynamic *IVS* selection at the edge, and the client adopts the default HLS setting which starts the playback from segment $N-2$, where N is the last fragment of the live playlist.

For comparison, we also provide the results for the *offline optimal* scheme, which makes an *offline* selection on an *IVS* with the highest average QoE (based on our former experiments) under the current network condition. It has the complete network throughput information and thus represents the best performance that *Rldish* and other methods can possibly achieve. To ease analysis, the performance data (i.e., the QoE of viewers) are collected at the edge directly using the methods introduced in § 3.5. It is worth mentioning that with Chrome DevTools, we validated that the QoE data collected at the browser side are pretty close to those collected at the edge.

3.6.3 QoE Criteria

There are well-defined metrics for the QoE of live viewers, which have been studied in [62, 63]. The QoE metrics considered in these work generally include startup latency, buffering ratio/time, playback delay (general latency), and rate of bitrate fluctuation. Since the bitrate decrease generally means QoE deterioration [67], this QoE deterioration can be roughly reflected by the total buffering time. Therefore, we consider startup latency (sl), total buffering time (bt), and general latency (gl) as defined in Section 3.4.3.

(1) QoE_{vs}: this QoE criterion favors low video buffering time by assigning a high weight to the live streaming with a low buffering time. It focuses on visual comfort of viewers. In our experiments, the weights of sl, gl, and bt in this QoE criterion are set to $\alpha = 10\%$, $\beta = 30\%$ and $\delta = 60\%$, respectively.

(2) QoE_{pg}: this QoE criterion favors low general latency (i.e., the progress of playback). It gives a high weight to the streaming with low general latency and thus focuses more on the timeliness of live streaming. Note that while QoE_{pg} values more on general latency, it does not necessarily mean that live viewers have to experience playback stalls during the watching process. Actually, the playback stalls can be avoided by adapting to a lower bitrate. In our experiments, the weights of sl, gl, and bt in this QoE criterion are set to $\alpha = 10\%$, $\beta = 60\%$ and $\delta = 30\%$, respectively.

Remark 2. *It is more illustrative to use a higher QoE score to mean better QoE. Since the lower values in startup latency, general latency, and total buffering time*

mean better QoE, we use the following equation to transform a metric to a (normalized and weighted) QoE score:

$$QoE(m) = 1 - w * \frac{m_{ob}}{m_{max}}, \tag{3.9}$$

where m denotes a QoE metric (i.e., startup latency, general latency, or total buffering time), m_{ob} is the observed value of m and m_{max} is the historical worst value of m, w denotes the weight of metric m. With this transform, a ***higher QoE score means a better performance****. All experimental results in the latter figures use the above transform.*

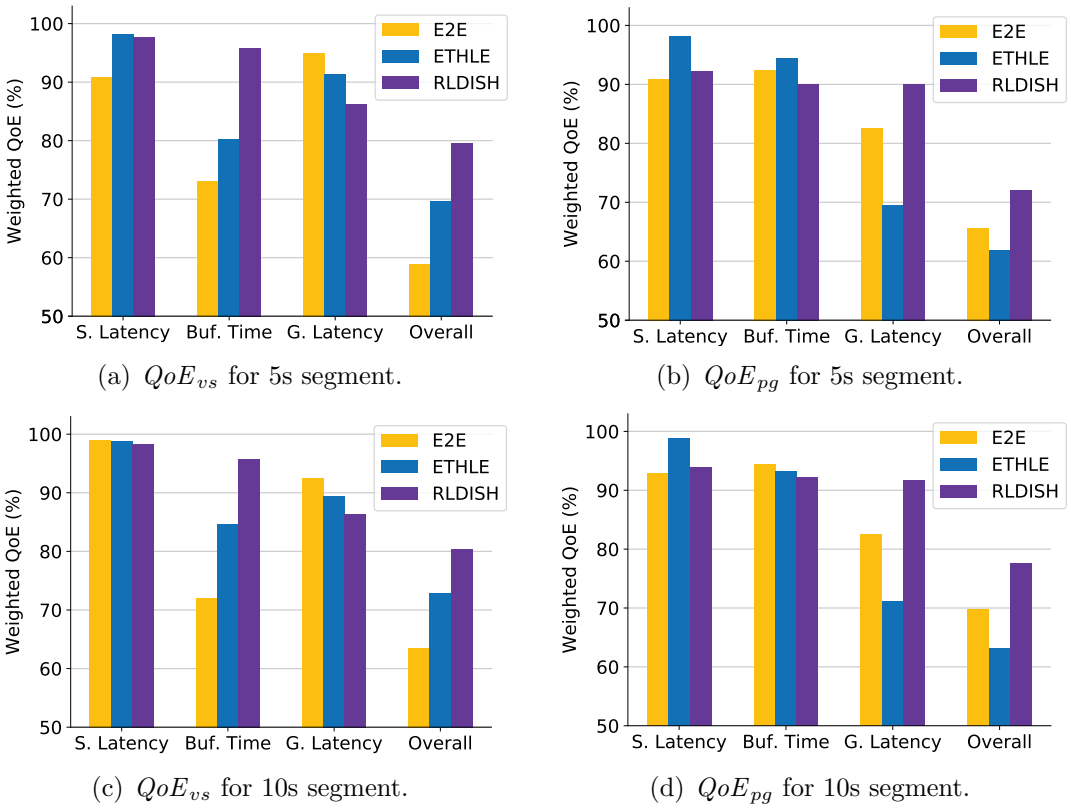
3.6.4 Performance Evaluation

Overall Performance

The average QoE for a certain segment length of each scheme (180 runs over 9 live streams, 1620 runs in total) is given in Fig. 3.6, where two QoE criteria QoE_{vs} and QoE_{pg} are considered. The figure shows the detailed QoE scores of all the schemes with the weighted score on each QoE metric. In the case of live stream with segments of 5 seconds, *Rldish* improves ETHLE by 14.2% and 16.5% w.r.t. QoE_{vs} and QoE_{pg}, respectively, and improves E2E by 35.9% and 9.8% w.r.t. QoE_{vs} and QoE_{pg}, respectively. In the case of live stream with segments of 10 seconds, *Rldish* improves ETHLE by 10.3% and 22.8% w.r.t. QoE_{vs} and QoE_{pg}, respectively, and improves E2E by 26.5% and 11.2% w.r.t. QoE_{vs} and QoE_{pg}, respectively.

Recall that QoE_{vs} favors low buffering time. *Rldish* under this QoE criterion sacrifices slightly on the general latency (by using a relative conservative IVS) in order to guarantee lower buffering time for the viewer. With QoE_{pg}, *Rldish* improves its performance on general latency while the performance on buffering time is degraded. The service provider can adjust the weights of different QoE metrics in the reward function to satisfy different QoE objectives.

Since ETHLE and E2E schemes could not adapt to different QoE objectives, their detailed QoE metrics (e.g., buffering time) should remain the same for the two QoE criteria cases. However, Fig. 3.6 shows a performance deterioration on QoE scores for the two schemes from QoE_{vs} to QoE_{pg}. This is because the QoE scores



(a) QoE_{vs} for 5s segment.

(b) QoE_{pg} for 5s segment.

(c) QoE_{vs} for 10s segment.

(d) QoE_{pg} for 10s segment.

Figure 3.6: The average performance on QoE of *Rldish* and other schemes for all live streams. The results are normalized and weighted based on the QoE criteria used. Refer to the error bars of Fig. 3.7 for the QoE distributions of *Rldish*.

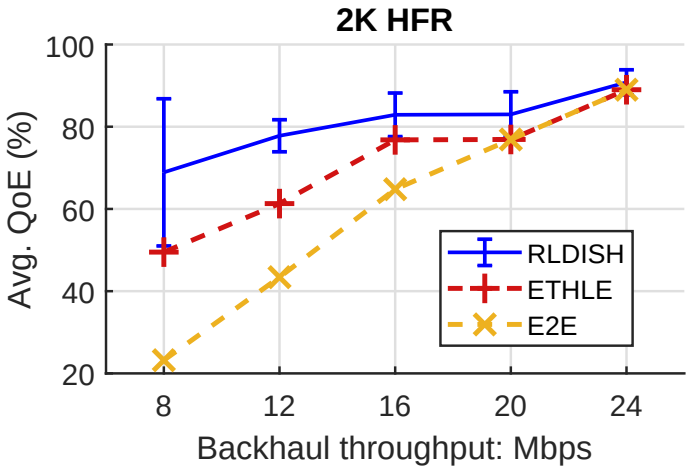


Figure 3.7: The average QoE_{vs} of 2K HFR live source under different network throughput using the dataset N.A. West VM as the streaming server

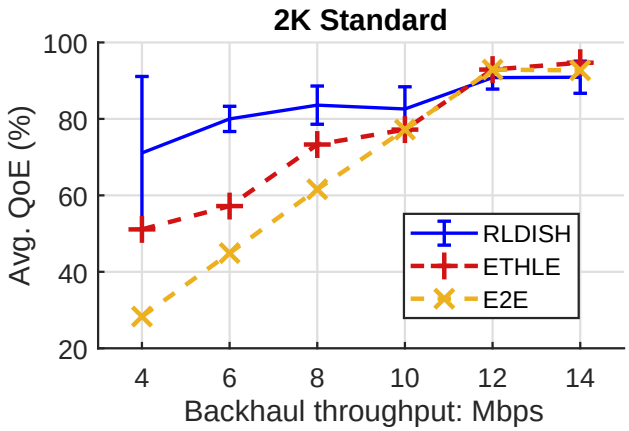


Figure 3.8: The average QoE_{vs} of 2K standard live source live source under different network throughput using the dataset N.A. West VM as the streaming server

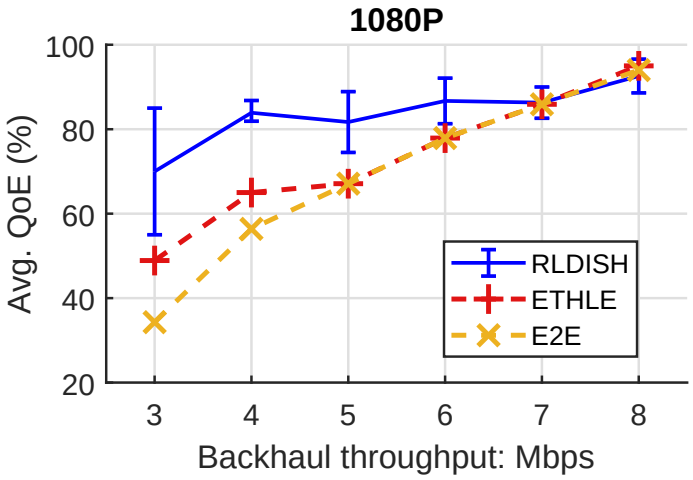
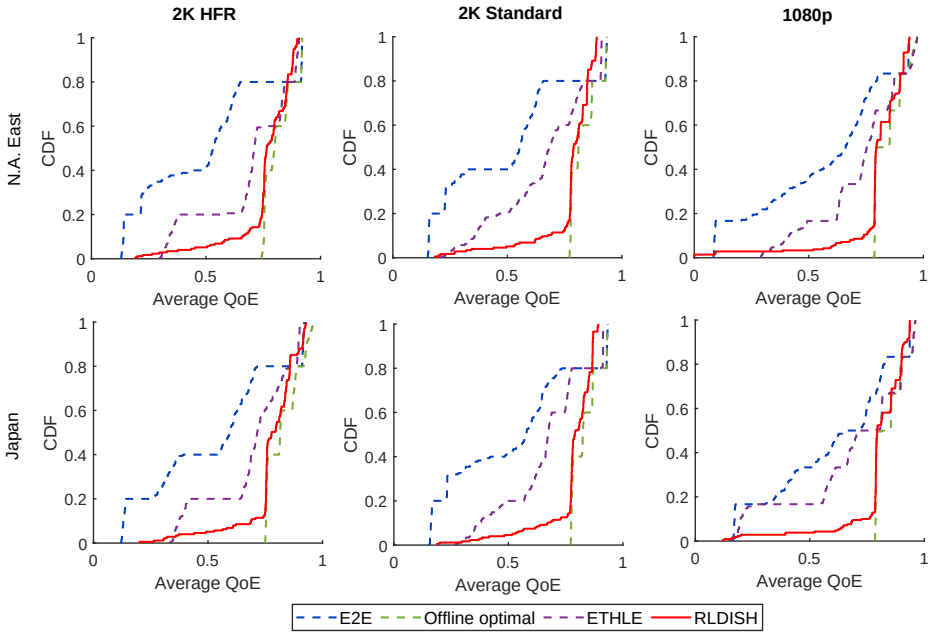


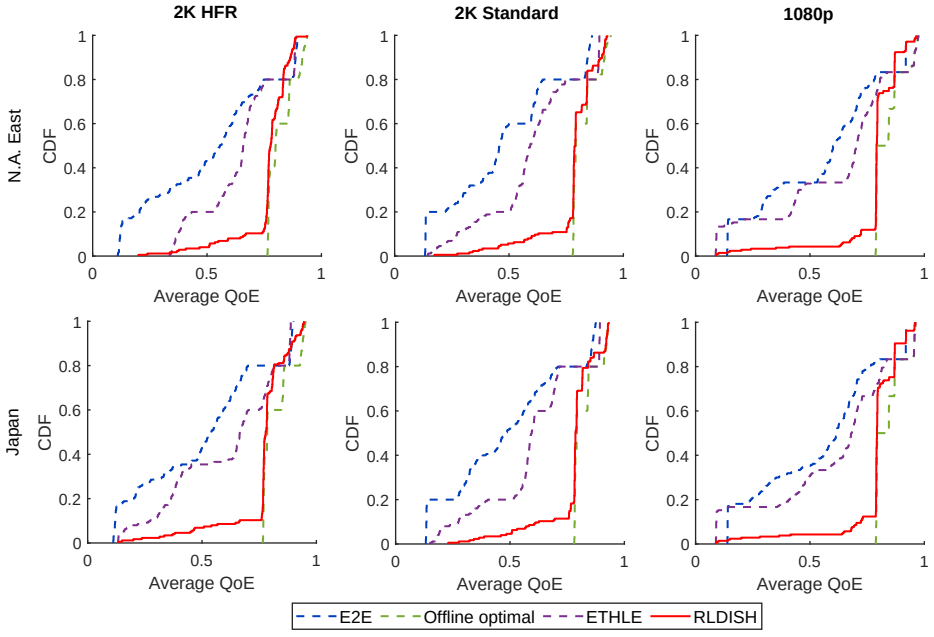
Figure 3.9: The average QoE_{vs} of 1080p live source under different network throughput using the dataset N.A. West VM as the streaming server

could get worse when the same latency value is multiplied by a higher weight (refer to (3.9)). Fig. 3.10 provides more detailed QoE_{vs} results in the form of CDFs for each live stream. Since the QoE results do not show too much difference between the streaming servers located in North America (N.A.) West and N.A. East, only N.A. East results are shown in this figure.

There are two key takeaways from Fig. 3.10. First, *Rldish* provides the majority of live viewers with the QoE score at around $75\% - 80\%$ (We can see a sharp increase during that range in the CDF). By conducting deep analysis on the QoE data from



(a) Stream of 5s segment



(b) Stream of 10s segment

Figure 3.10: CDF results of QoE_{vs} of live viewers for *Rldish* and other schemes with streaming severs located in North America and Japan respectively.

our experiments, we find that these QoE values are mainly contributed by the *IVS* selections at a critical point: these are the *IVS* choices which reach a good balance in buffering time and general latency for the viewer. At the critical point, *Rldish*

should not use a newer segment as the *IVS* since that will lead to the punishments on playback stalls. These *IVS* selections normally lead to small buffering time (close to zero) and similar general latency. Since they are usually the newest video segment in the edge server (*IVS* just hits the edge cache), their startup latency is also pretty small.

Second, *Rldish* may generate a small fraction of poor decisions. It has overall around 5% of total viewer's QoE in the range of 20% to 50% according to CDF. These poor decisions are mainly caused by the exploration choices of RL including the startup explorations. This problem could be alleviated, since startup explorations of RL could be accomplished by pre-testing clients (machines) located near the edge server. Fig. 3.7, 3.8 and 3.9 breaks down the QoE scores of all schemes under different throughput of backhaul network. Despite a relatively high standard deviation on QoE scores during the startup phase of RL (shown with error bars), *Rldish* outperforms the other schemes when backhaul network throughput is low.

3.7 Conclusions

In this chapter, we tackled the technical challenges in applying RL to *IVS* selection of HTTP-based live streaming to improve the QoE of live viewers. Compared with the state-of-the-art solutions, *Rldish* is lightweight and completely transparent to both the clients and the streaming server. We deployed *Rldish* as a virtualized network function (VNF) in a real HTTP cache server, and performed extensive real-world experiments to evaluate its performance. In the following chapter, we will propose an alternative solution to improve the QoE of Edge-assisted live video delivery.

Chapter 4

Proactive Content Replication for Edge-Assisted Live Video Delivery

In this chapter, we will propose an alternative solution aiming at improving the QoE of edge-assisted live video delivery. Our solution use a proactive live live video push scheme to resolve the cache miss problem in live video delivery.

4.1 Introduction

The last few years have witnessed the dramatic proliferation of live video streams over streaming platforms (such as Twitch, Facebook Live, and Youtube Live, etc.), which have generated a revenue of billions of dollars [57]. According to Twitch, in 2019, over 660 billion minutes of live streams were watched by customers, and 3.64 million streamers (monthly average) broadcast their channels via Twitch [68].

Nevertheless, the delivery of live videos is quite different from the conventional video-on-demand (VoD) service. First, live video has quite spiky traffic, which means the viewer popularity of live streams usually grows and drops very rapidly [10]. In particular, it often encounters the "thundering herd" problem [11, 12]: a large number of users, sometimes on the scale of millions, may start to watch the same live video simultaneously when some popular events or online celebrities start a live broadcast. Second, live video delivery nowadays has stringent latency requirements due to the new breed of live video services that support interactive live video streaming. These services allow the broadcasters to interact with their stream viewers in real-time during the streaming process. In order to support the high interactivity, it requires

low-latency end-to-end delivery while maintaining the Quality of Experience (QoE) for live viewers [13, 14, 15].

Typical thundering herd problem in the live video can overload the system, causing lags and disconnections from the server. One efficient way to solve the thundering herd problem while maintaining low latency in live videos is to utilize edge caches. For example, Facebook uses edge PoPs distributed worldwide to deliver their live traffic [11]. Providing contents via the edge (e.g., CDN edge servers, crowdsourced edge devices [16]) makes contents much closer to the end users and alleviates the traffic burden of backbone networks to the cloud.

Nevertheless, when applying edge-assisted live video delivery, a new problem of cache miss arises: when a large number of end users request for a newly generated video segment at the same time, this segment may not have enough time to be cached in the edge caches due to the real-time property of live streaming [17, 18]. As shown in Fig. 4.1, the edge server would return a cache miss for the first group of requests that arrive at the edge before the segment is fully cached. These cache-missed requests would pass the edge cache and go all the way to the origin cache or server. As a result, it would lead to deteriorated QoE to the live viewers (e.g., increased startup latency and playback stall rates). According to Facebook [11], around 1.8% of their Facebook Live requests encountered cache miss at the edge layer. Note that 1.8% is a significant number considering the large number of total live viewers. To make matters worse, high revolution videos (e.g., virtual reality (VR) streams) need more time to be replicated to the edge and would create an even higher cache miss rate.

The above caching problem only exists for live video streaming as people typically watch regular videos at different times. Therefore, there is sufficient time for the regular video chunks to be cached with few fast-entered content requests. State-of-the-art research solves the above problem by holding back the availability of some newly encoded live segments from the playback clients so that the client requests could arrive at the edge after the caching process has finished [11, 18]. While this strategy solves the cache miss problem, it may pose extra delays to the live streams [69], which compromise the "liveness" of delivered videos.

The root cause of the cache miss problem is mainly because the current client-driven caching strategy was not designed for live videos in the first place. Since caching process in the current content delivery networks (CDNs) is normally triggered by the client requests, the video segments caching (replication) will only commence when the cloud responds to the first request for a video segment. While this strategy

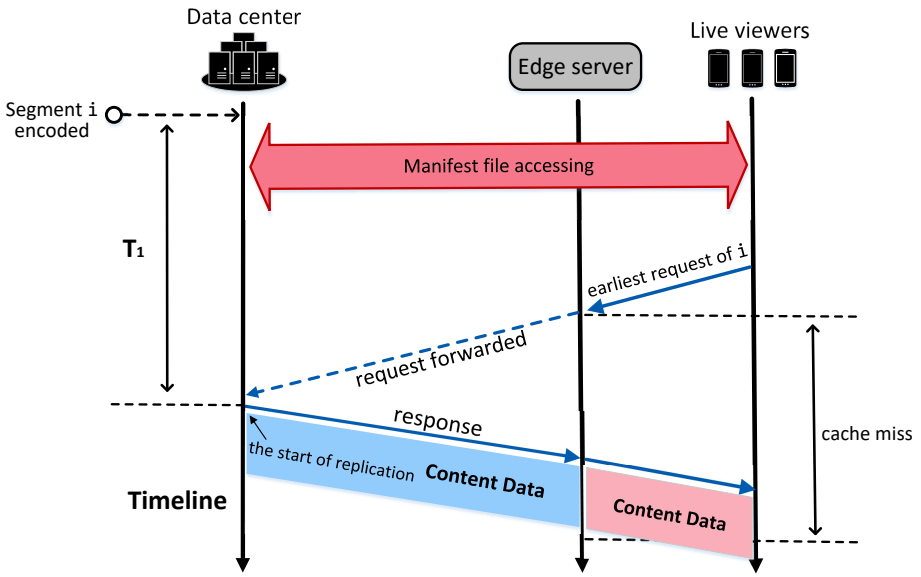


Figure 4.1: Client-driven content caching (replication) for live videos.

makes sense when delivering regular content, it slows down the caching process in the context of live videos: there exists a time *gap* (shown as T_1 in Fig. 4.1) between the time when a segment is generated from the cloud and when the caching process starts. This gap mainly consists of two parts: i) the time for the the playback clients to obtain the availability information of the newly encoded video segments, and ii) the time it takes for the clients to send their first segment request. However, in the current pull-based CDN architecture, both times are difficult to narrow down (refer to § 4.2 for more details). This motivates us to rethink the caching design of live video delivery. Can the cloud CDN server adopts a video push model to proactively replicate the newly encoded video segments into the appropriate edge servers such the video replication could commence before the user requests?

Although desirable, it is challenging to achieve such a goal due to the massive video requests and edge servers, QoE guarantee, and stringent real-time requirement. First, to adopt the proactive caching strategy, we must solve the allocation problem between edge servers and live viewers (i.e., assign the viewers to the proper edge server). This is because i) the video segments that need to be replicated in an edge server are determined by the live viewers served by this edge server, and ii) as the bandwidth capacity of edge servers is quite limited (much smaller than CDN servers), some edge servers could be heavily loaded while the others are under-utilized. Conventional load

balancing solutions [70, 71] assume that content replicas are stored over *all* CDN servers. This assumption is unrealistic in our context due to the massive number of edge servers. Second, since the service capability of each individual edge server is limited, newly encoded video segments should be replicated to multiple edge servers to alleviate the spiky live video traffic. For each live video segment being encoded in real-time from the cloud, we need to make a fast decision on the appropriate edge servers to cache the segment.

In this chapter, we propose a proactive live video edge replication scheme (*PLVER*) to resolve the cache miss problem in live video delivery. *PLVER* first conducts a *one-to-multiple* stable allocation between edge clusters and user groups to balance the load of live requests over edge servers. In this way, each user group is assigned to its most preferred edge cluster whenever possible. Based on the allocation result, *PLVER* then proposes an efficient proactive live video edge replication (push) algorithm to speed up the edge replication process by using real-time statistical viewership of the user groups allocated to a cluster.

In summary, we makes the following contributions:

- *PLVER* implements a stable *one-to-multiple* allocation between edge clusters and user groups (i.e., one user group is served by one edge cluster but one edge cluster can serve multiple user groups), under the constraint that the QoE of end users is guaranteed by their assigned edge clusters.

- Aiming at speeding up the edge replication process, *PLVER* identifies the unique traffic demand of live videos and develops a proactive video replication algorithm to provide fast and fine-grained replication schedule periodically. To the best of our knowledge, this is the first research work to provide proactive video replication algorithms (with details disclosed to the public) tailored for edge-assisted live video delivery.

- We perform comprehensive experiments to evaluate the performance of *PLVER*. Trace-driven allocations between 641 edge clusters and 1253 user groups are conducted, covering 64 ISP providers and 470 cities. Based on the allocation results, we further evaluate the performance of the video replication algorithm using traces of 0.3 million Twitch viewers and more than 300 Twitch channels. Performance results demonstrate the superiority of *PLVER*.

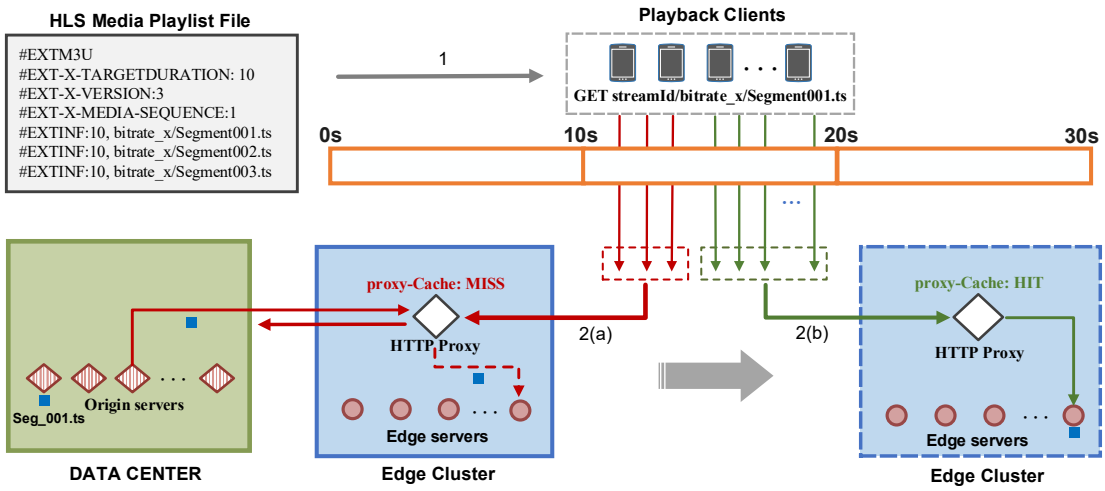


Figure 4.2: Illustration of cache miss problem for edge-assisted live video delivery.

4.2 Related Work

4.2.1 Live Video Delivery Background

A live stream is usually encoded into multiple pre-determined bitrates once it is generated and uploaded by the broadcasters. For each bitrate, the stream is further split into a sequence of small video segments with the same playback length. The stream can be fetched sequentially by playback clients (e.g., via `HTTP GET`), using a suitable bitrate matching their network conditions [27].

In the HTTP-based live video delivery, every time when a client joins a live channel, she first requests and accesses the stream's playlist file (generated by the origin streaming server). This manifest contains the information of currently available segments (i.e., segments that have been encoded in the cloud) and the bitrates in the stream. Based on the manifest information, the clients send the HTTP requests to their local edge server. Afterwards, the playback client fetches the live video segments in sequence and periodically accesses the newest playlist file to check if any new segments have been produced. When live videos are delivered over edge servers (as shown in Fig. 4.1), these video segments will be replicated (cached) to the edge caches when the edge HTTP proxy receives the response (segment) from the cloud.

4.2.2 Observation and Motivation

To better explain the cache miss problem, we use Apple HLS (HTTP Live Streaming) protocol [28] as an example to illustrate the live video delivery process. As shown in Fig. 4.2, starting from a certain time after 10^{th} second of a live stream, the first three video segments were generated from the cloud. By first accessing the playlist file, clients (with geographical proximity) realize the segment update and begin to request segment 001.ts via `HTTP GET` during 10 to 20 seconds. These requests would first be handled by one of the HTTP proxies in an edge cluster, which checks if the requested segment is already in an edge cache. If the segment is in the edge cache, it could be readily fetched from there (step 2(b)). If not, the proxy will issue a HTTP request to the origin server in the cloud (step 2(a)). (Note that there exists another layer of cache, proxy and encoding servers inside the data center. As our system design does not change the current structure within the data center, these components are omitted in Fig. 4.2.)

We can easily find that an earlier fraction of requests before the segment is fully cached in the edge (shown as step 2(a) in Fig. 4.2) would miss the edge cache. The current client-driven caching architecture creates a time gap before the caching process is started, which is critical for the live video delivery with real-time requirement. The gap mainly consists of two parts. The first part contains the time that the playback clients request and access the playlist file, which is inevitable in the client-driven content caching since the clients have to know the segment information (i.e., the URI) before sending the requests. Once the playback clients obtain the video segment availability information, the clients need another period of time before sending the first request for the segment. This part of time exists because the current live streaming protocols (e.g., HLS or MPEG-DASH) generally start live streaming with a relatively "older" video segment instead of the newest one to avoid playback stalls [28]. As shown in Fig. 4.2, the playback clients would start the live streaming by first requesting segment 001.ts rather than segment 003.ts, making the replication of segment 003.ts further postponed in the client-driven caching architecture.

4.2.3 Improving the QoE of Live Streaming

In order to solve the cache miss problem as well as to improve the QoE of 4K live videos, Ge et al. [18] proposed ETHLE, which "holds back" the availability of some newly encoded video segments from the playback clients so that the playback clients

could send their live requests to a certain segment after it was cached in the edge server. While this work has shown considerable QoE improvement, it may pose extra undesirable latency to the live streams.

In the industry, Facebook proposed two alternative methods to solve the cache miss problem for delivering live video over edge servers [11]. In the first scheme, their solution uses a similar "holding back" idea as that in [18]: the edge proxy returns a cache miss for the first request while holding the rest requests in a queue. Once the segment is stored in the edge cache via the HTTP response for the first request, the requests in the queue can be responded from the edge as cache hits. Similar to the work in [18], this design would incur undesirable latency to the live stream. The other scheme adopts a video push model where the server continuously pushes newly generated video segments to the proxies and the playback clients. This is the only reported design that adopts proactive content push for live video over edge servers. Nevertheless, the exact details of their video replication algorithm are unknown.

In [72], Yan et al. proposed LiveJack, a network service that allows CDN servers to leverage the ISP edge cloud resources to handle the dynamic live video traffic. Their work mainly focuses on the dynamic scheduling of Virtual Media Functions (VMFs) at the edge clouds to accommodate the dynamic viewer populations. Wang et al. proposed an edge-assisted crowdcast framework, which makes smart decisions on viewer scheduling and video transcoding to support *personalized* QoE demands [57]. Mukerjee et al. [2] performed end-to-end optimization of live video delivery path, which coordinates the delivery paths for higher average bitrate and lower delivery cost. This work, however, mainly focuses on optimizing the routing of live video delivery. In [73], Zhang et al. provided a video push mechanism to lower the bandwidth consumption of CDN by proactively sending the videos to competent seeds in a hybrid CDN-P2P VoD system. This work uses proactive video push, but it does not target live videos. In [69], Ray et al. proposed a live-streaming upload solution that improves the overall QoE by considering the unique character of live video that live streams are typically watched by live viewers at different delays. The main focus of this work, however, is to optimize the bandwidth usage during the video upload process (i.e., from broadcasters to the cloud) instead of the video replication from the cloud to the edge servers. The optimization for regular, non-live videos delivery was also investigated in [74, 75] and [76].

4.2.4 Generic Video Replication Techniques

Different content replication strategies were developed in [77, 16, 78, 79] and [80]. In [77], Hu et al. considered both video replication and request routing for social videos. Their algorithm focuses on social videos and the watching interests of different communities. In [16], Ma et al. considered the video replication strategies in edge servers. They proposed a content replication algorithm to jointly minimize the accumulated user latency and the content replication cost. In [78], Zhou et al. investigated how the popularity of video changes over time and then designed the video replication strategies with the video popularity dynamics derived. In [79], Applegate et al. presented an approach for intelligent video placement that can optimize the bandwidth usage for the IPTV VoD services. Different from ours, the above works mainly focus on the video-on-demand (VoD) services.

4.3 System Overview

Our system design of *PLVER* is shown in Fig. 4.3. Once a live viewer sends a `HTTP` request to the *request manager* of the system, the request manager identifies the key information of the request, including the requested channel, bitrates, and the user group it belongs to, by resolving the `URL` and the `IP` address. The above information is used by the request manager to redirect the request to an appropriate edge server. This procedure is denoted with blue-dash lines in Fig. 4.3. The request manager also generates the viewership information (e.g., the number of viewers of each stream in each user group) and feeds the information to *PLVER* for edge servers selection.

As shown in Fig. 4.3, there are three main components *PLVER*: i) *stable allocation* module assigns the global user groups to their desired edge server cluster and balance the load of live traffic, ii) the *proactive replication algorithm* periodically computes the edge replication schedule within each edge cluster in the near future (e.g., next 5 minutes), based on the viewership information from the request manager, and iii) *replication table* which contains the directly available information of replication servers for each live video segment.

When the new live video segments of a stream are encoded and generated, the system first checks the most up-to-date replication schedule from the replication table by identifying the key information of the segment. It then proactively replicates these video segments into the guided edge servers despite that these videos are currently

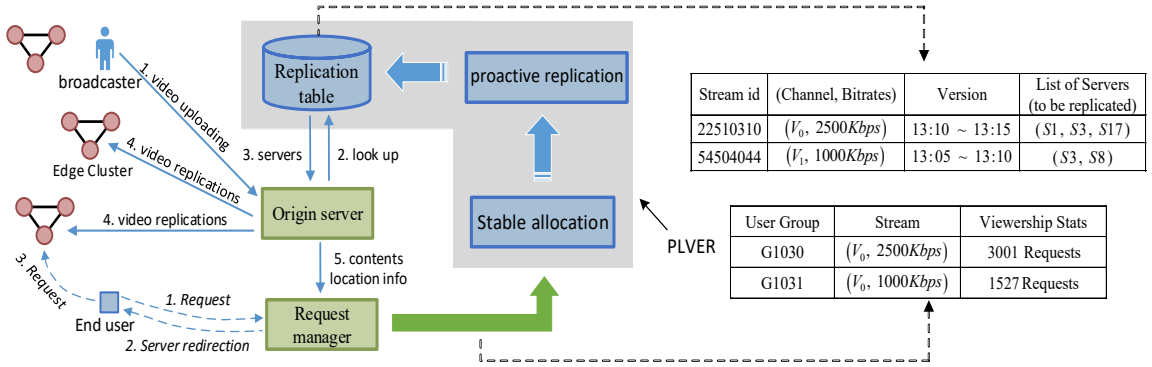


Figure 4.3: System architecture: solid lines denote the procedure for video replication; dash lines denote the procedure that a user accesses live video.

not requested by the users. In this way, the replication schedule can be obtained easily by using the stream id and version number of the target video segment as the key for searching. Note that the process of replicating the video segments into edge servers and delivering the video contents from edge servers to the end users are conducted concurrently, since the video segments are generated from the broadcasters sequentially.

The core component of the system is *PLVER*, denoted in the grey box in Fig. 4.3. Its main goal is to provide a replication schedule that can be readily used for live video replication over edge servers. To be more specific, it considers the traffic demand from different areas as well as the resource capacity of edge servers so as to provide the replication schedule that maximizes the traffic served by the edges. Note that the task of tracking each viewer's requests and directing the requests to edge servers or the origin server belongs to real-time request redirection. It happens after the replication schedule is generated and needs to consider the dynamic content availability in edge servers. It is thus beyond the scope of this chapter. Nevertheless, it will be utilized for the performance evaluation of our algorithm in the evaluation part (§ 4.7) of this chapter.

In the following, we formally model the problem that needs to be solved by *PLVER*, and then present the two main components of our solution, namely *stable one-to-multiple allocation* and *proactive replication algorithm*, in § 4.4 and § 4.5, respectively.

4.4 Stable One-to-multiple Allocation

4.4.1 The Allocation Problem

Instead of making the request routing decisions individually for each client, we conducted the servers allocation at the granularity of user groups. The users in the same group generally have the same network features (*e.g., subnet, ISP, locality*) and thus are likely to experience similar QoE (e.g., startup latency and video buffering ratio) when dispatched to the same server [29, 30]. Similar to conventional content delivery problems, it is generally necessary to first consider the load balance problem between edge server clusters (consisting of a number of edge servers with the same network features) and the user groups.

We consider a target network of a number of edge server clusters and user groups. Each user group i originates an associate live traffic demand d_i, and each edge cluster j has a capacity C_j to serve the demands. In order to satisfy the QoE of users, for each user group, it has a list of candidate edge clusters in descending order of preference. A higher preference indicates those clusters that can provide better-predicted performance for the viewers in the group (e.g., lower latency and packet loss). Likewise, each edge cluster j also has preferences regarding which map units it would like to serve [81].

An allocation of edge clusters to user groups is said to be a *stable marriage* if there is no pair of participants (i.e., edge clusters and user groups) that both would be individually better off than they are with the element to which they are currently matched [82]. By conducting stable allocation, each user group is assigned to its most preferred server cluster to which it could be assigned in any stable marriage. In other words, stable allocation implies the most desirable matching between user groups and server clusters. The goal of our allocation problem is to assign the user groups to the edge clusters, such that the capacity constraints are met and the bidirectional preferences are accounted for.

4.4.2 Stable Allocation Implementation Challenges

However, in the context of live video delivered over edge servers, the stable allocation has practical implementation challenges listed below.

Expensive many-to-many assignment

Conventional allocation used by CDN vendors normally generates a many-to-many assignment, i.e., the traffic demand from each map unit could be served by several (multiple) edge clusters that are geographically near the map unit. Many-to-many assignment makes sense when there are only a small number of server clusters globally. However, In our context, the number of edge clusters is much more than that of conventional CDN clusters. A many-to-many assignment under this situation becomes unnecessarily expensive. For example, given that there are tens of thousands of edge clusters, it is expensive to measure the performance of all the edge clusters and split the traffic demand from a single user group to too many edge clusters. Note the traffic demand generated by a user group in our context is also much smaller than the demand generated by a conventional map unit in conventional CDN.

Partial preference lists

Considering a large number of edge clusters and user groups, it is unnecessary to measure and rank the preference of every edge cluster for each user group. Therefore, there is a partial preference list of edge clusters that are likely to provide the best performance for a given user group. Similarly, the edge clusters also only need to express their preferences for the top user groups that are likely candidates for the assignment.

Integral demands and capacities

The canonical implementation of stable marriage problem considers unit value demands and capacity, while in our case the demands of user groups as well as the capacities of server clusters could be arbitrary positive integers.

4.4.3 Solution Methodology

To address the above challenges, we propose a new allocation algorithm: *Integral Stable One-to-multiple Allocation (ISOA)*, which extends the Gale-Shapley algorithm used for solving the canonical stable allocation problem. *ISOA* works in rounds. In each round, each free user group (all user groups are free initially) proposes to its most preferred edge cluster, and the edge cluster could (provisionally) accept the proposal. Let G_i denote the list of user groups assigned to edge cluster i. In the case that the

Algorithm 3: *ISOA*

Input: Preference list by user group and edge cluster: uP, cP; C_j: resource capacity of an arbitrary edge cluster j; $D()$: traffic demand of given user groups.

Output: G_j: List of allocated user groups to edge cluster j.

```
1  Initialize all user groups as free; G_j = {j : [], for j in E};
2  foreach i ∈ free user groups do
3      j ← head of uP_i;
4      Insert i into G_j (rely on cP_j);
5      if D(G_j) <= C_j then
6          continue;
7      start ← bSearch(G_j, C_j, i); k ← start + 1;
8      while k ≤ length(G_j) do
9          if D(G_j^{0~k}) > C_j then
10             remove G_j^k from G_j;
11             if G_j^k == i then
12                 remove j from uP_i; goto line 3;
13             else
14                 label G_j^k as free user groups;
15         k ← k + 1;
16 return G;
```

Algorithm 4: bSearch(G_j, C_j, i)

```
1  left ← position of i in G_j; right ← length of G_j;
2  mid = (left+right)/2;
3  while True do
4      if D(G_j^{0~mid}) ≤ C_j then
5          if D(mid + 1) > C_j then
6              return mid;
7          else
8              mid = (mid+right)/2;
9      else
10         mid = (left+mid)/2;
```

capacity of the edge cluster is violated, we perform a binary search on G_i to identify the user groups that need to be evicted.

Algorithm 3 shows the details of *ISOA*, where uP and cP are the preferred list of

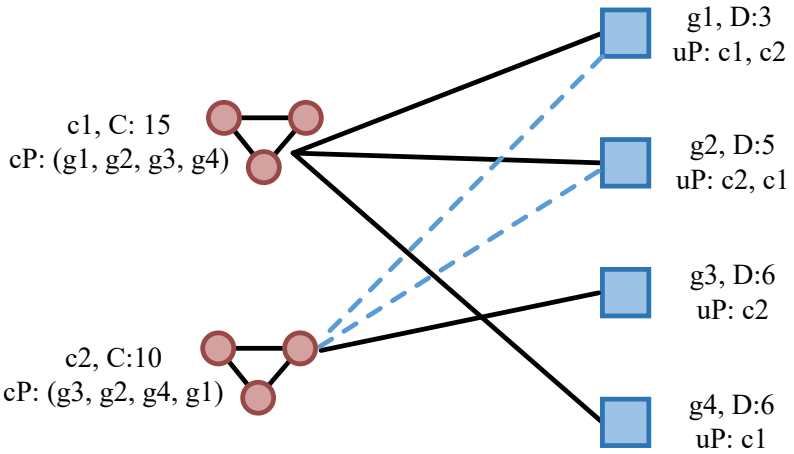


Figure 4.4: An example of the stable one-to-multiple allocation containing four user groups and two edge clusters, where the service capacity of each edge cluster is denoted by 'c' and the traffic demand of each user group is denoted by 'D'

edge clusters (by user groups) and the preferred list of user groups (by edge clusters), respectively. C_j denotes the service capacity of edge cluster j (Note that in practice, C_j could be a *resource tree* instead of a single value [81]). To find a stable one-to-multiple allocation, we first set all user groups as free and set the initial user groups to each edge cluster as empty (line 1). Then, we pick up a free user group i in each round and get its most preferred edge cluster j (line 2-3). Based on the preference of edge cluster, we insert i into the temporarily-assigned user group list of edge cluster j (i.e., user groups in G_j are sorted according to cP_j). After adding a new user group to G_j, the current traffic demand needed by G_j may or may not violate the capacity C_j. If C_j is not violated, we go back to propose another free user group for proposing (line 5-6).

In case of C_j is violated, we conduct a binary search (refer to Algorithm 4) to find out the first user group $(start + 1)$ in G_j which causes the capacity violation. We further go through all user groups from $G_j^{start+1}$ to the end of G_j: if adding a user group (G_j^k) would cause the capacity violation, then we remove this user group from G_j (line 8-10). If the removed user group is i itself, it suggests that i cannot get its most preferred edge cluster. In that case, i will go back to propose to its second preferred edge cluster (line 11-12). Otherwise, the evicted G_j^k will be labelled as a free user group, waiting for a second-chance proposal.

As a simple example, Fig. 4.4 shows the meaning of the stable one-to-multiple

Table 4.1: Summary of main notations in § 4.5

Notation	*Description*
F	Target edge cluster within which the replication problem to be solved
E	Set of all edge servers over F
U	Online viewers from the user groups assigned to F
T	Target time window in the near future
A_j	Set of viewers that are served by edge server j
B_j	Bandwidth capacity of edge server j
a_i	The edge server that serve viewer i during T
b_i	Bandwidth consumed by viewer i
s_i	The live stream that viewer i is watching
c_j	Cache capacity of edge server j.
$\mathcal{D}_i^T$	The video segments to be generated by s_i in T
L_{A_j}	Non-redundant set of streams accessed by viewers in A_j
$S(\cdot)$	Size function that calculates the total data volume in a set of video segments
$\mathcal{V}_j^T$	Live video segments that should be replicated into edge server j during T
$L(v_i^t)$	The replication schedule: list of edge servers that video segments v_i^t should be replicated into

allocation, where we have two edge clusters $c1$ and $c2$, with a service capacity of 15 and 10, respectively. There are 4 user groups ($g1$, $g2$, $g3$ and $g4$), which generate traffic demands of 3, 5, 6 and 6, respectively. The preferred edge cluster list by each user group as well as the preferred user group list by edge cluster are shown in this figure with uP and cP, respectively. We need to match each user group to their most preferred edge cluster that it could be assigned to. Running *ISOA* with the simple example in Fig. 4.4, user group $g1$, $g3$, $g4$ can propose and be matched to their most preferred edge cluster ($c1$, $c2$ and $c1$, respectively). Group $g2$, however, will trigger the capacity violation of $c2$, thus can only be matched to its second preferred cluster $c1$. The matching results are marked with the solid lines in Fig. 4.4.

Let N denote the number of user groups in the target network. The time complexity of *ISOA* is $O(N * log(N))$. This is based on the fact that i) the number of edge clusters is way less than the number of user groups, and ii) the algorithm consists of N rounds of proposals and the bottleneck within each round is on the binary search and insert. Typically, the computation of the ISOA algorithm should be performed at a regular interval (e.g., per day or even per week).

4.5 Proactive Replication over the Edge

4.5.1 Notations and Assumptions

Once the allocation problem is solved, we only need to consider the replication problem within each single edge cluster and its assigned user groups (Note that the QoE of the assigned user groups is guaranteed with stable allocation). Next, we formulate the single cluster replication problem by considering a given edge cluster F and the user groups assigned to F. The main notations used in our problem formulation are listed in Table 4.1. Without loss of generality, we make the following assumptions:

- We consider a target time window T in the near future that we need to generate the video replication schedule. During T, a number of live streams are watched by the live viewers U distributed across the user groups allocated to F.
- Each end user is served by one edge server at a time, and clients that cannot be served by the edge servers will be directed to the cloud.
- The cache in an edge server can be shared by multiple viewers who are accessing the video, but each viewer consumes their exclusive bandwidth of the edge server.

4.5.2 Resource Constraints

We divide time into a series of short, consecutive time windows, and try to generate a video replication schedule for each time window based on the feed of most up-to-date viewership of live videos. The goal of the replication schedule is to maximize the traffic served by the edge servers so as to improve the QoE of end users.

Since clients generally access the video segments of a live stream sequentially, users' demands to the video segments to be generated in the next short time window can be roughly estimated by the current viewership of this stream. Note that the live viewers might change their video quality (bitrates) during the watching process, while the distributed design and a fine-grained time window allow the system to quickly respond to stream demand change. We use a_i to represent the edge server that serves viewer i in T, and use A_j to denote the set of viewers served by server j, i.e.,

$$A_j := \{i | a_i = j, \forall i \in U\}. \tag{4.1}$$

Since each viewer only needs to be served by one edge server, we have

$$A_{j_1} \cap A_{j_2} = \varnothing, \forall j_1 \neq j_2. \tag{4.2}$$

For an arbitrary edge server j, it should have enough bandwidth to serve A_j. Thus, the following constraint should be posed:

$$\sum_{i \in A_j} b_i \leq B_j, \forall j \in E, \tag{4.3}$$

where b_i is bandwidth consumed by viewer i, and B_j is the total bandwidth of edge server j.

Let s_i denote an arbitrary live stream of one live channel with a certain bitrate. We denote the video segments to be generated by s_i in T as $\mathcal{D}_i^T$ (i.e., $\mathcal{D}_i^T = \{v_i^{t_1}, v_i^{t_2}, \ldots, v_i^{t_n}\}$, where $(t_1, t_2, \ldots, t_n)$ are the timestamps of the video segments $(v_i, v_i, \ldots, v_i)$ generated in T, respectively). If we use L_{A_j} to denote the non-redundant set of streams accessed by viewers in A_j (note that $|L_{A_j}| \leq |A_j|$ as one stream is normally watched by more than one viewers), the following constraint on cache capacity should be posed:

$$\sum_{i \in L_{A_j}} S(\mathcal{D}_i^T) \leq c_j, \forall j \in E, \tag{4.4}$$

where $S(\cdot)$ is the function that calculates the total caching size of a given set of video segments, c_j denotes the cache capacity of edge server j.

4.5.3 Cost of Content Replication

While edge servers benefit the live viewers, we may need to generate multiple replicas of single video content over the edge servers. More replicas on the edge servers generally mean more cost of cache resources at the edge as well as extra delivery cost from the cloud to the edge servers.

To reach a good balance, we pose the following constraint to limit the overall replication cost:

$$\sum_{j \in E} \sum_{i \in L_{A_j}} S(\mathcal{D}_i^T) \leq \alpha \cdot \sum_{j \in E} c_j, \tag{4.5}$$

where $\sum_{j \in E} \sum_{i \in L_{A_j}} S(\mathcal{D}_i^T)$ is the total size of overall replicas cached in the edge servers, and $\sum_{j \in E} c_j$ represents the total size of videos that could be cached globally.

We use α, a percentage variable, to bound the total amount of videos that could be replicated, so as to limit the video replication cost.

4.5.4 Problem Formulation

The problem needs to be solved by our proactive replication algorithm can be formulated as:

$$\max_{\{a_i, A_j\}} \quad \sum_{j \in E} \sum_{i \in A_j} b_i \tag{4.6a}$$

$$\text{s.t.} \quad (4.2), (4.3), (4.4), \ \textit{and} \ (4.5) \tag{4.6b}$$

Solving (4.6), we obtain a_i and A_j, with which the video replication schedule could be easily derived. That is, the video segments that should be replicated into edge server j during T are given by the following:

$$\mathcal{V}_j^T = \sum_{i \in L_{A_j}} \mathcal{D}_i^T. \tag{4.7}$$

Based on $\mathcal{V}_j^T$ of each edge server, we can do a simple reverse transformation to get the video replication schedule, i.e., for an arbitrary video segment from stream i with timestamp t (v_i^t), the list of edge servers to which it should be replicated in T is given by:

$$L(v_i^t) = \{j | v_i^t \in \mathcal{V}_j^T, \forall j \in E, \forall t \in T\}. \tag{4.8}$$

This video replication schedule is then inserted into the *Replication table* in Fig. 4.3, by identifying the channel and bitrate of stream i. Problem (4.6) is an integer linear program with a massive number of design variables. In the rest of this section, we present a two-step heuristic algorithm to solve this problem.

4.5.5 Solution Methodology

Since live video traffic is network intensive [81], the non-sharable bandwidth constraint is a harder constraint compared with the sharable cache capacity constraint. Therefore, *PLVER* first considers the replication problem while temporarily ignoring the constraint on the cache capacity (**Step 1**). It then conducts adjustments by moving workloads from the edge servers where the cache capacity constraint is violated

to the edge servers with available cache and bandwidth resources (**Step 2**).

Step 1: Greedy Edge Replication for Maximum Traffic: By temporarily ignoring the cache capacity constraint, the replication problem could be transformed into the *Multiple Knapsack Problem (MKP)* [83]. This problem is defined as a pair $(\mathcal{B}, \mathcal{S})$ where $\mathcal{B}$ is a set of m bins and $\mathcal{S}$ is a set of n items. Each bin $j \in \mathcal{B}$ has a capacity c_j, and each item i has a weight w_i and a profit p_i. The objective is to assign the items to the bins such that the total profit of the assigned items is maximized, and the total weight assigned to each bin does not exceed the corresponding capacity.

If we treat the profit of each viewer i as the bandwidth consumption b_i of that viewer, the *MKP* problem is equivalent to our replication problem with unlimited cache capacities, i.e.,

$$\max_{\{x_{ij}\}} \quad \sum_{j\in\mathcal{B}}\sum_{i\in\mathcal{S}} b_i x_{ij}, \tag{4.9a}$$

$$\text{s.t.} \quad \sum_{i\in\mathcal{S}} b_i x_{ij} \leq B_j, \forall j \in \mathcal{B} \tag{4.9b}$$

$$\sum_{j\in\mathcal{B}} x_{ij} \leq 1, \forall i \in \mathcal{S} \tag{4.9c}$$

$$x_{ij} \in \{0,1\}, \forall i \in \mathcal{S}, j \in \mathcal{B}, \tag{4.9d}$$

where $\mathcal{B}$ and $\mathcal{S}$ are defined as the set of edge servers in a given edge cluster and the set of viewers assigned by the stable one-to-multiple allocation, respectively, and x_{ij} is defined as follows:

$$x_{ij} := \begin{cases} 1 \text{ , if viewer } i \text{ is served by edge server } j, \\ 0 \text{ , otherwise.} \end{cases} \tag{4.10}$$

The *MKP* problem has been well-researched and has a polynomial-time approximation solution (*PTAS*) [83]. Once solving Problem (4.9), we could further calculate the set of video segments that should be replicated into each edge server based on the value of x_{ij}. Let $\mathcal{P}_j$ denote the set of video segments that should be stored in edge server j after Step 1.

Step 2: Workload Adjustment: The solution $\mathcal{P}_j$ can maximize the total amount of traffic served by edge cluster under the assumption of unlimited cache capacities of edge servers. Posing the constraint of limited cache capacity, we need to further adjust the solutions by moving part of the replication workloads from the

Algorithm 5: *proactive replication algorithm*

Input: $\mathcal{F}$: given edge cluster; cc_j and bd_j: available cache capacity and bandwidth of edge server j ($j \in \mathcal{F}$), respectively; $\mathcal{M}$: set of user groups allocated to $\mathcal{F}$; s_i: the live stream accessing by viewer i; b_i: bandwidth consumption of s_i;

Output: Replication schedule $\mathcal{S}_j$, i.e., for each edge server j, the set of video segments should be replicated during the near time window T.

```
/* Phase 1: Generate initial replication schedule by solving MKP. */
x_ij ← solvingMKP(bd_j, b_i); S_j ← ∅ , ∀j ∈ F
a_i = ∅, for all viewer i ∈ M;
foreach edgeServer j ∈ F do
    I = {i|x_ij = 1} //viewers that are served by server j;
    L ← the set of unique streams consumed by the viewers in I;
    sort L according to R(s, j); (refer to (4.11))
    foreach stream s ∈ L do
        if D_s^T.size() > cc_j then
            continue;
        S_j.append(D_s^T); cc_j ← cc_j − D_s^T.size();
        a_i = j, (for all i ∈ I and s_i = s); update bd_j;
/* Phase 2: Redirecting viewers. */
foreach viewer i with a_i = ∅ do
    if exists server j ∈ F with bd_j ≥ b_i && s_i ∈ S_j then
        a_i ← j; bd_j ← (bd_j − b_i);
/* Phase 3: Offloading replication tasks. */
while exits edge server j with available resources do
    if a_i ≠ ∅ for all i ∈ M then
        break;
    pick the stream s with max R(s, j) and cc_j ≥ D_s^T.size();
    if s ≠ ∅ then
        S_j.append(D_s^T); cc_j ← cc_j − D_s.size();
        bd_j ← bd_j − R(s, j); φ = R(s, j)/b;
        update φ (number) unassigned viewers with a_i = j, ∀i with s_i = s;
return S_j;
```

edge servers whose cache capacity is violated, to the edge servers that have spare cache and bandwidth.

The whole proactive replication algorithm is shown in Algorithm 5, including three phases. Phase 1 represents Step 1, and phases 2 and 3 represent Step 2 introduced

above. Once phase 1 is finished, there might be some viewers whose demand cannot be satisfied (i.e., $a_i = \varnothing$) if we pose cache capacity constraint. For each of these unassigned viewers, in phase 2 we try to redirect it to an edge server that has available bandwidth capacity and has the required video segments cached already. There are no directly available edge servers that could be used to serve the rest of the unassigned viewers after phase 2. Thus, in phase 3, the algorithm offloads the incomplete video caching tasks to the edge servers with residual resources. The algorithm returns when all traffic demands are completely satisfied or all edge servers in the given edge cluster are fully loaded.

The time complexity of our algorithm is $O(n + m)$ (n and m are the number of viewers in $\mathcal{M}$ and the number of edge servers in F, respectively), without considering the first step of solving the *MKP* problem. Since there are different approximation schemes for solving *MKP* in polynomial time and *PLVER* is a decentralized algorithm with viewers and edge servers from a single edge cluster (i.e., n and m in a small magnitude), *PLVER* could be solved easily.

Remark 3. *Note that PLVER does not need to track the real-time information of each viewer (e.g., stream being watched, bandwidth consumption). Instead, it only needs the statistics on the number of viewers of each stream (viewership) in each user group. In this sense, a "viewer" in PLVER actually means the corresponding resource demand to each live stream.*

Remark 4. *We introduce the reward for caching a stream s in a certain edge server j (line 7 in Algorithm 5). It implies the traffic demand that could be served by caching the video segments of this stream in server j (during T). Let b denote the bitrate of stream s; then the reward of s could be defined as following:*

$$\mathcal{R}(s, j) = b * min\{\lfloor \frac{\bar{B}_j}{b} \rfloor, N\}, \tag{4.11}$$

where $\bar{B}_j$ is the current available bandwidth of edge server j, and N is the number of viewers on stream s that have not been assigned to a server (i.e., $a_i = \varnothing$). The reward is in accord with our objective (4.6a), i.e., maximizing the amount of traffic served by edge servers.

Remark 5. *The PLVER algorithm is to solve the optimization problem (6). This problem is the multidimensional (i.e., multiple constraints) multiple knapsack problem (MMKP) [84]. According to [85], even for the multidimensional one knapsack*

problem, no fully polynomial approximation scheme exists as soon as the number of constraints is greater than or equal to 2*. Similarly, for a* 1*-dimensional multiple knapsack problem, no fully polynomial approximation scheme exists if the number of knapsacks is greater than or equal to* 2 *[83]. To the best of our knowledge, there is no algorithm that solves MMKP with a proven constant approximate ratio. The PLVER algorithm solves the problem by first considering the major constraint (i.e., bandwidth constraint) for multiple knapsacks and then "repairing" the solution if the secondary constraint (i.e., cache capacity) is violated. In practice, after the harder bandwidth constraint has been satisfied, the cache capacity constraint may only require some minor adjustment of the existing allocation, i.e., redirecting and offloading. The redirecting does not impact the value of the objective function, and only offloading (Step* 3*) reduces the value of the objective function. The impact of the offloading step is not big, as shown in the later evaluation in Section 3.6. Thus in practice, PLVER should have an approximation ratio close to that of the algorithm presented in [83] for solving the multiple knapsack problem, i.e.,* $(2+\epsilon)$*-approximation.*

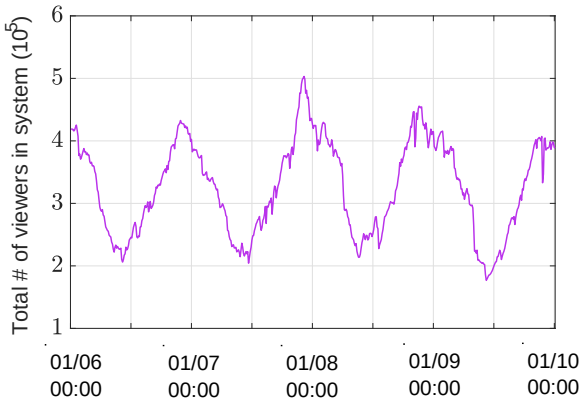
4.6 Experimental Setup

We conduct *trace-driven* simulations in Python and Matlab to evaluate the performance of our ISOA and PLVER algorithms. The simulation is conducted using real-world live video dataset and the ISP coverage information.

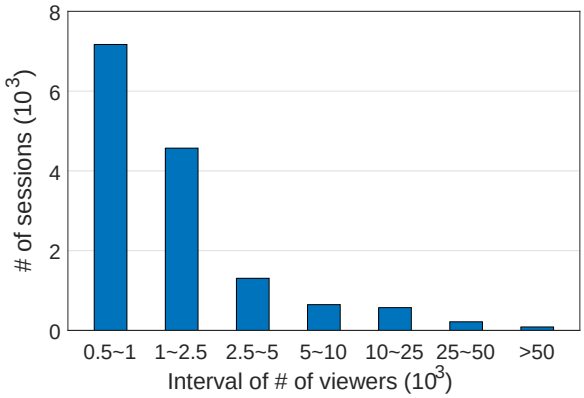
4.6.1 Live Video Viewership Dataset

Twitch provides developers with a RESTful API to obtain the live video information. In our experiment, we use a public dataset [86] that consists of the traces of thousands of live streaming sessions on Twitch [87]. The dataset contains the information of all live channels in the Twitch system, with a sampling interval of 5 minutes. Detailed information includes the number of viewers of each channel, bitrates of each channel, and the duration of live sessions. We select the live channels that have more than 100 viewers and extract the required information for these channels.

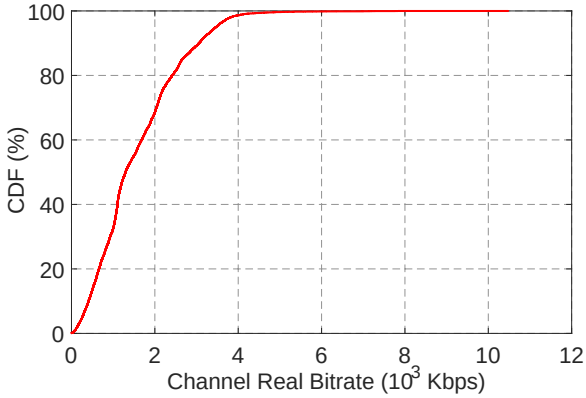
Fig. 4.5(a) shows the total number of viewers in the system from Jan. 06 to Jan. 09. During a certain time period, a channel can be either *online*, which means that it is broadcasting a live video, or offline. When a channel is online, we say that it corresponds to a *session*. Fig. 4.5(b) shows the distribution of sessions with the



(a) Number of viewers in the system over time.



(b) The distribution of number of sessions with different number of viewers.



(c) The CDF of the bitrates of live channels.

Figure 4.5: The statistical information of the experimental dataset.

different average number of viewers.

Fig. 4.5(c) illustrates the distribution of bitrates of channels in the dataset. Based on the video encoding guidelines [88], we assume that the video streams can be encoded with multiple standard resolutions (or bitrates): 240p, 360p, 480p and 720p

Table 4.2: Different levels of preferred edge cluster by user groups.

Preference Priority	**Clusters Description**
Lv. 1	clusters that are within the same ISP and located in the same city.
Lv. 2	clusters that are within the same ISP and located in the same county.
Lv. 3	clusters that are located in the same city while with different ISPs.
Lv. 4	clusters that are within the same ISP and located in the same state.
Lv. 5	clusters that are located in the same county while with different ISPs.
Lv. 6	clusters that are located in the same state while with different ISPs.

(or $400, 750, 1000, 2500$ *Kbps*). Obviously, while a channel broadcasts with bitrate b, the viewers of this channel cannot select the video quality with a bitrate exceeding b.

4.6.2 Target Network & User Groups

geoISP [89] collected the detailed performance and region coverage information of $2,317$ Internet Service Providers (ISPs) in the US. Based on the information, we build a target network over two US states (Washington and Oregon). We further develop a web crawler to collect the ISP coverage information of 470 cities over 70 counties in the two states from the website of geoISP.

We divide all the viewers from these two states (about 0.3 million on average) into 1253 user groups (based on the combination of ISP and city) within our target network. Note that one ISP can cover multiple cities and one city can be covered by multiple ISPs. From the dataset, we know the percentage of users in a city that is supported by a particular ISP. For each live stream, we distribute its viewers among these user groups based on the population of each user group (calculated based on each city's population and the ISP coverage percentage of the city).

4.6.3 Edge Server Clusters

Setup of edge clusters & servers

Among all user groups, we further extract 641 city-ISP combinations as the target for deploying the edge clusters. Each edge cluster in our experiments consists of five types of edge servers with 5, 10, 20, 40 and 80 Mpbs bandwidth capacity, respectively. The servers are randomly deployed at each edge cluster. The total bandwidth of all the deployed edge clusters is set to equal the total traffic demand of all viewers. Note that such bandwidth setting may not always guarantee the full satisfaction of all viewers' demand because the bandwidth of each edge cluster may not be fully utilized and also because the cache capacity and QoE of each edge cluster may be different. Nevertheless, our later experiment shows that such a setting is appropriate to evaluate the performance of different edge caching strategies.

Setting of cache capacity

For an edge server with bandwidth capacity b Mbps, it should have at least $b * T$ Mb (T is the considered time period) cache capacity to ensure that it has enough resources in our edge replication strategy. For simplicity, we use $\hat{b}$ to denote $b * T$ hereafter. Since video traffic delivery is network intensive, the cache capacity of edge servers is normally larger than $\hat{b}$ Mb. In our experiments, we assume that the cache capacity (variable X) of all edge servers is uniformed distributed within the range of $(0.5 * \hat{b}, 2 * \hat{b})$. In the following section, we will further adjust the capacity that could be used in each edge server by setting different values of replication cost constraint factor α.

4.7 Performance Evaluation

4.7.1 Performance Evaluation of Stable One-to-multiple Allocation

Evaluation Methodology

We compare the performance of *ISOA* with another edge cluster allocation strategy: *greedy allocation*. With the greedy allocation, each of the user groups selects its most

Table 4.3: Allocation results of ISOA.

Preference rank of the allocated cluster	# of user groups at each preference level		
	Greedy Allocation	ISOA	Changes
Lv.1	390	451	+61
Lv.2	496	457	-39
Lv.3	120	106	-14
Lv.4	136	127	-9
Lv.5	56	57	+1
Lv.6	36	37	+1
un-allocated	19	18	-1

preferred edge cluster iteratively, until all user groups get allocated or there is no available edge cluster.

Preference List Generation

We define the rank of preferred edge clusters of user groups (introduced in § 4.4), as listed in Table 4.2. Note that the preference list defined in Table 4.2 is just an example paradigm to generate the input of our stable allocation algorithm, and it can be altered by the CDNs themselves, e.g., according to the contract terms under which the cluster is deployed, the granularity of the user groups partition, and so on [81].

Performance Evaluation of ISOA

We conduct the stable one-to-multiple allocation between user groups and edge clusters based on the data introduced in 4.6.2 and 4.6.3, using *ISOA* and the aforementioned *greedy allocation* method. The detailed allocation results are summarized in Table 4.3. Since there are 1253 user groups in our experiment, the table shows the distribution of these user groups being allocated with different levels of preferred edge cluster. For example, there are 390 user groups that are allocated with their first ranked (most preferred) edge cluster with *greedy allocation*, while the number is increased to 451 with *ISOA*. Compared with the greedy allocation, *ISOA* can allocate more user groups with their higher-ranked (i.e., more-preferred) edge clusters.

The average preference rank of the allocated edge clusters of each user group over

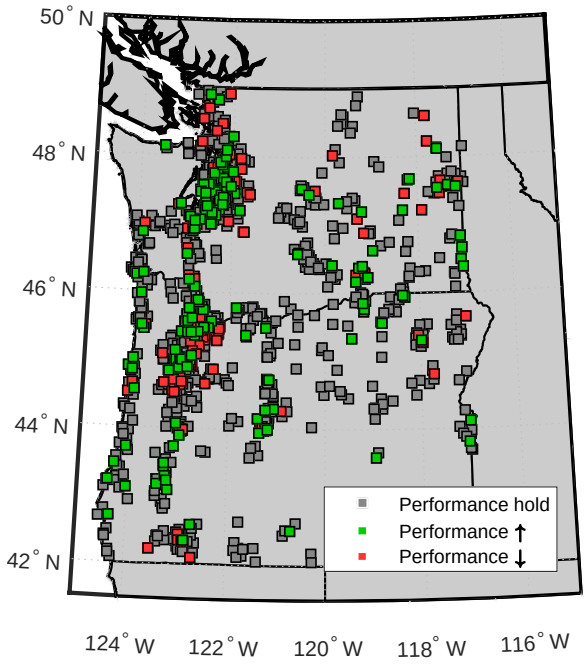


Figure 4.6: Performance change with ISOA over *greedy allocation*.

the whole network is 2.18 and 2.25 for *ISOA* and *greedy allocation*, respectively, and the variance is 1.72 and 1.67 for *ISOA* and *greedy allocation*, respectively. Over the 468 cities which include user groups, the average preference rank of the allocated edge clusters is 2.40 and 2.44 for *ISOA* and *greedy allocation*, respectively, and the variance is 1.62 and 1.53 for *ISOA* and *greedy allocation*, respectively.

The performance improvement with *ISOA* over the greedy allocation for every user group in our experiment is further illustrated in Fig. 4.6, where the performance of each user group is marked with a colored square.

4.7.2 Performance Evaluation of Proactive Edge Replication

Evaluation Methodology

We evaluate *PLVER* by comparing it with the following replication strategies:

- *Auction Based Replication (ABR):* Each edge server conducts a simple "auction" to determine the cached videos: live videos with the largest number of viewers via the edge server win the auction and are cached, and the auction repeats until the edge server uses up its cache capacity [90]. In other words, this method replicates the videos into an edge server based on their current

number of viewers in the decreasing order.

- *Caching On Requested Time (CORT): This strategy does NOT adopt pre-replication and using request triggered caching strategy instead.* It caches the videos into the edge servers in real-time when video segments are truly requested by the end users. When content requested, it first checks if there are available edge servers to serve this request; if not, it replicates the video segments of this stream into a new edge server.

To evaluate the performance of different strategies, we use the metric ***offloading ratio***, which is calculated by the amount of traffic served by the edge servers divided by that of the overall traffic in the time period of length T. The performance is evaluated under different values of ***replication cost factor*** α (refer to § 4.5.3), so that we can investigate the tradeoff between performance and replication overhead.

Overall Performance of PLVER

Based on the twitch viewership data from Jan. 06, 2014 to Jan. 09, 2014, we conduct experiments on an hourly base. By setting the value of α to $20\%, 40\%, 60\%, 80\%$ and 100%, we compute the average offloading ratios of the three strategies in each case. The results are shown in Fig. 4.7, from which we can see that *PLVER* outperforms *ABR* and *CORT* in all five cases. The overall performance improvement by *PLVER* for the five cases are $9\%, 10\%, 15\%, 28\%$ and 10% over *ABR*, respectively, and $82\%, 82\%, 79\%, 81\%$ and 44% over *CORT*, respectively.

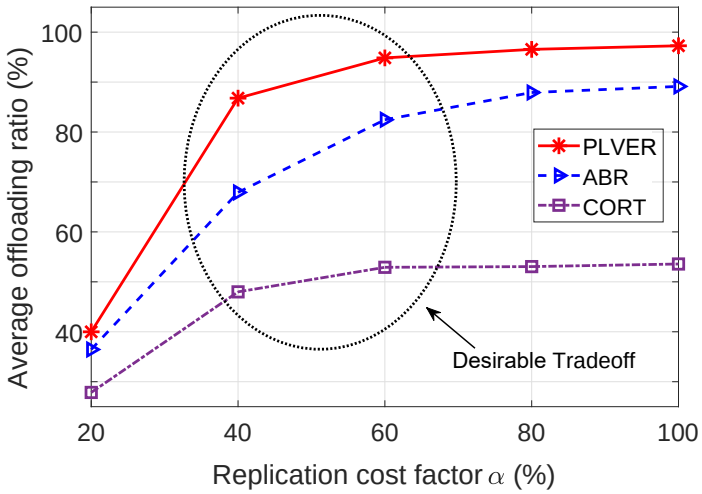


Figure 4.7: Avg. traffic offloading ratio with different α.

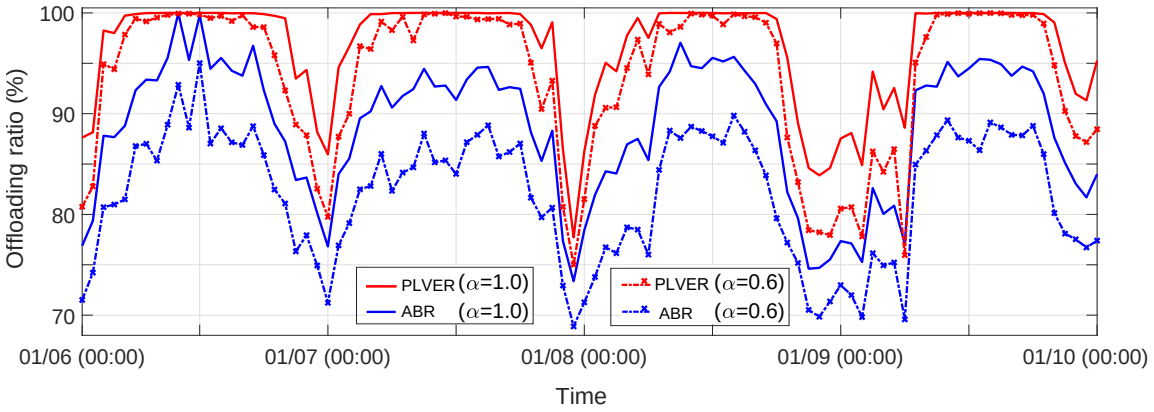


Figure 4.8: The hourly performance of *PLVER* and ABR.]

Furthermore, we can find from Fig. 4.7 that the overall performance got a considerable improvement when the replication cost constraint (α) is increased from 20% to 60%. However, the performance improvement fades when α continues to increase after 60%. This situation holds for all the three replication strategies. Therefore, in our experiment, it reaches a good tradeoff between performance and replication costs when α is between 40% and 60% (as shown in Fig. 4.7).

Detailed Performance of PLVER

Referring to the overall performance, *ABR* is more comparable to *PLVER* (than *CORT*). We thus investigate the detailed performance behaviors of *PLVER* and *ABR*. The traffic offloading ratios for i) each hour and ii) each user group are shown in Fig. 4.8 and Fig. 4.9, respectively.

Fig. 4.8 shows the hourly traffic offloading ratio of *PLVER* and *ABR* with the replication cost constraint factor α equal to 100% and 60%, respectively. It shows that even within non-peak hours (when the resources of edge servers are sufficient), it is hard for *ABR* to yield a satisfying performance. In contrast, when α decreases from 100% to 60%, the performance degradation of *PLVER* is much smaller than that of *ABR*.

Fig. 4.9 shows a heat map indicating the performance of *PLVER* and *ABR* at each edge cluster (with $\alpha = 40\%$), where the traffic offloading ratios are represented by different colors. Since there are no edge clusters with performance less than 30% or greater than 90%, our color bar denotes the traffic offloading ratio from 30% to 90%. An edge cluster with better performance is colored in green, and worse in red.

Under this setting, the average traffic offloading ratio of each edge cluster over the target network is 65% and 78% for *ABR* and *PLVER*, respectively, and the variance is 0.006 and 0.004 for *ABR* and *PLVER*, respectively.

We also investigate the performance when requesting different video qualities (240p, 360p, 480p, 720p). The satisfaction ratio of requests (i.e., the ratio of requests that are successfully directed to corresponding edge servers) with different replication strategies is shown in Fig. 4.10(a). We can observe that *PLVER* outperforms *ABR* and *CORT* for all types of quality requests. Among the four different quality requests, the high-quality request of 720p has a relatively low traffic offloading ratio than the other three video qualities. However, as high-quality requests generate more traffic than the others, it impacts more on the final performance. *PLVER* provides a satisfaction ratio of 36% for the 720p requests, which is higher than those of *ABR* (19%) and *CORT* (30%), respectively.

Impact of Viewership Fluctuation

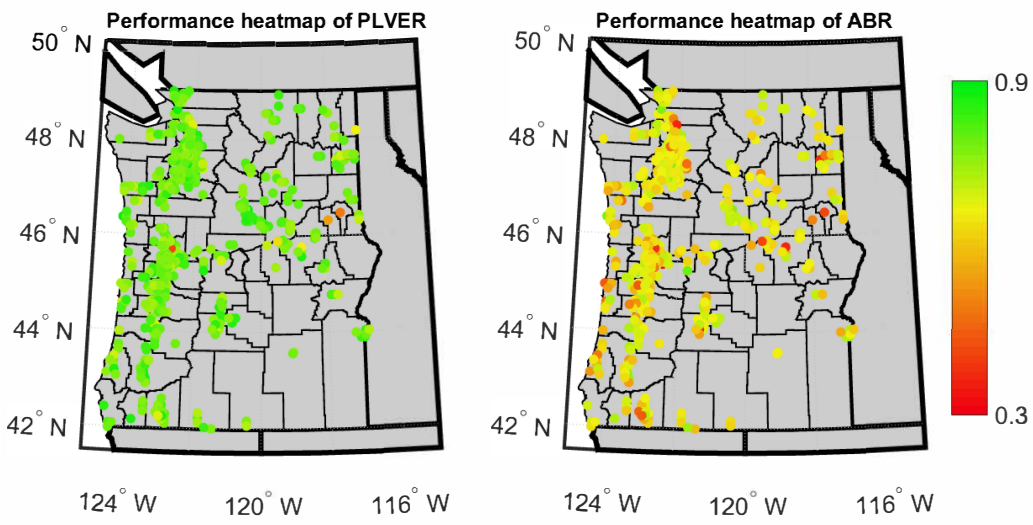


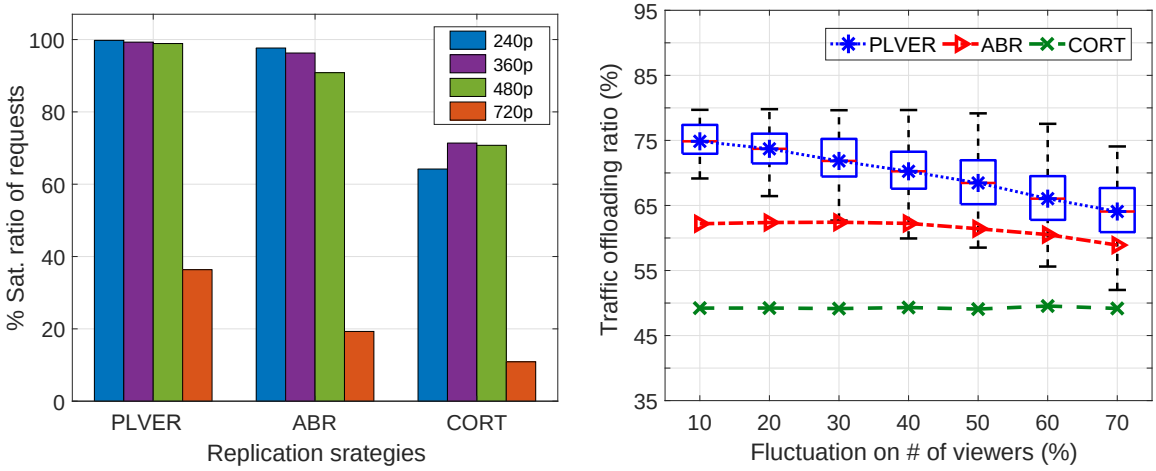
Figure 4.9: The performance of *PLVER* and ABR in each edge cluster with $\alpha = 0.4$.

As *PLVER* makes use of the viewership information (i.e., the number of viewers) in the current time window to make decisions in the next time window, the viewership fluctuation in consecutive time slots may impact the performance of replication algorithms. To investigate that, we first generate the replication schedules by different replication strategies referring to the viewership data in peak traffic hours of § 4.6.1, then we manually generate a new viewership data to test the performance of these

replication schedules. The new viewership data is generated by introducing different levels of fluctuations on the former viewership data that we used to generate the replication schedules. To be more specific, the number of viewers of each channel are added with different percentages of fluctuation (e.g., randomly plus or minus 20%).

The performance of *PLVER* under different viewership fluctuations is shown in Fig. 4.10(b). We can see that the performance curve (representing the traffic offloading ratios) slightly goes down from 75% to 64% with fluctuations changing from 10% to 70%. Nevertheless, according to the statistical analysis of our dataset, viewership fluctuations higher than 30% are quite rare. Hence, its impact on *PLVER* is quite small.

Efficiency Analysis



(a) The satisfaction ratio for video requests with different qualities.

(b) The variation of performance with the fluctuation change on the number of stream viewers.

Figure 4.10: Performance results of *PLVER* and other replication strategies.

We evaluated the processing performance of the edge replication algorithm of *PLVER* on a desktop computer without specific performance optimizations. The desktop ran Windows 10 with an Intel©Core™i7-2600 CPU with 4GB of memory. The average processing time of *PLVER* is 0.47 seconds on the desktop. Typically, the computation of edge replication schedule should be performed every time period (e.g., one minute) during which the live stream viewership has been changed (to a certain extent). Considering that the duration of a single video segment is normally between 2-10 seconds [18, 28], a replication schedule would normally be able to provide the

replication guidance (i.e., the target edge servers to replicate a live stream) for the next 10 (approximately) video segments of a live stream.

It is worth mentioning that, our experiments show that the workload adjustment of Algorithm 5 does not lead to big changes to the replication guidance derived from solving the initial MKP problem. With the replication cost factor α set as 60%, each edge server could fulfill 82.5% (on average) of original video replication tasks (derived from solving MKP) at phase 1 without requiring workload adjustment. When α is changed to 80%, which represents a more abundant cache capacity, this ratio is increased to 93.5%.

4.8 Conclusion

Live video services have gained extreme popularity in recent years. The QoE of live videos, however, suffers from the cache miss problem occurred in the edge layer. Solutions from the current live video products as well as the state-of-the-art research would pose extra latency to the live streams which sacrifices the "liveness" of the delivered video. In this chapter, we propose *PLVER*, an efficient edge-assisted live video delivery scheme aiming at improving the QoE of live videos. *PLVER* first conducts a *one-to-multiple* stable allocation between edge clusters and user groups. Then it adopts proactive video replication algorithms over the edge servers to speed up the video replication over edge servers. Trace-driven experimental results demonstrate that our solution outperforms other edge replication methods.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this dissertation, we investigated three critical problems for the QoE enhancement of live videos over CDNs.

First, in order to improve the live video delivery performance under current multi-CDN strategies, we proposed a novel and feasible solution to multi-CDN, termed as CDN semi-federation. Our solution can better schedule and utilize the resources from multiple CDNs without requiring full CDNI. We used an effective optimization algorithm which reshapes the patterns of traffic from multiple CPs and delivered over multiple CDN Points of Presence (PoPs). Experiments across North American and European ISP PoP networks demonstrate that, compared with current multi-CDN solutions, CDN semie-federation can reduce the content delivery latency by around 20% during peak traffic hours.

Second, to optimize the QoE of HTTP-based live videos, we proposed *RLdish*, a scheme deployed on the edge CDN server, which can dynamically select a suitable *IVS* for new live viewers based on Reinforcement Learning (RL). Our solution makes the IVS decisions on a per-stream basis which avoids the high overhead in the current throughput estimation based solution. *RLdish* is lightweight and transparent to both the client and the streaming server. It collects the real-time QoE observations from the edge without any client-side assistance. We evaluate the performance of our solution using streaming servers distributed around the world. Our experimental results show that *RLdish* improves the QoE of HTTP-based live streaming services by up to 22% w.r.t. QoE measures such as buffering time and latency.

Last but not least, aiming at solving the cache miss problem in live video delivery, we proposed *PLVER*, a proactive live video push scheme to further enhance the QoE of live videos delivered over edge CDN servers. Our solution first conducts a one-to-multiple stable allocation between edge clusters and user groups, then adopts proactive video replication algorithms to speed up the process of video replication over edge CDN servers. Compared with other state-of-the-art solutions, *PLVER* can greatly maintain the liveness of the delivered videos. We conduct extensive trace-driven evaluations, covering 0.3 million Twitch viewers and more than 300 Twitch channels. The experimental results demonstrate the superiority of *PLVER*.

5.2 Future Work

The line of this thesis research was originally motivated and partially sponsored by industry (Ericsson Canada). During the pandemic period, real-time live video streaming, video conferencing, and online teaching are booming. This trend may continue in the near future. As such, we expect that current CDNs will be faced with unprecedented challenges, demanding people to revisit current CDN technology and develop novel solutions. This dissertation fits this need and can be extended from several directions.

First, in this dissertation, we improved the QoE of HTTP-based live streaming services by customize the *IVS* selection of different live viewers based on the network condition. While HTTP-based live streaming services (e.g., HLS) are widely adopted nowadays, it would normally pose extra latency to the delivered videos since the size of video segments in HLS are relatively large (normally between 2 and 10 secs). In a lot of scenarios, it is important that live viewers can have near real-time conversations without an data transmission delay. Real-time messaging protocol (RTMP) is also a widely used streaming protocol which maintains a persistent TCP connection between the user and the server during the whole broadcast [11]. Since each chunk of RTMP protocol in only 64 ms long, it helps to further bring down the latency. In the future work, the research problems such as how to improve the QoE of RTMP-based streaming services, need further investigation.

Second, in Chapter 2, we evaluate our multi-CDN content delivery strategy based on the assumption that a PoP is able to serve any types of traffic request so long as the performance requirement (e.g., latency) could be met. However, due to the limited cache capacity of CDN PoPs as well as the content storage cost, in practice,

a certain PoP does not necessarily need to store all the content replicas from all CPs. Therefore, in the future work, it would be necessary to first solve the replica placement problem (i.e., which are the content replicas that need to be cached at a certain time t for a certain PoP k?) by jointly considering the time-varying traffic demands as well as the constraint of content storage cost. Then based on the dynamic content placement, we could further verify the performance of our solution.

Third, our edge replication algorithms in Chapter 4 rely on the viewership information of the end users. This method may induce performance degradation when there exist high fluctuations on the real-time viewership. In fact, the trend for the number of viewers of a certain stream is usually predictable, e.g., during the short time period when a broadcaster just gets online or at the beginning of peak traffic hours, the number of viewers of a stream is usually increasing. In the future work, we could model the viewership patterns of the live streams and predict viewership such that the algorithms could be fed with more precise information.

Appendix A

List of Publications from the Thesis

1. H. Wang, G. Tang, K. Wu, and J. Fan, "Speeding up multi-cdn content delivery via traffic demand reshaping," in *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 422–433

2. H. Wang, K. Wu, J. Wang, and G. Tang, "Rldish: Edge-assisted qoe optimization of http live streaming with reinforcement learning," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2020, pp. 706–715

3. H. Wang, G. Tang, K. Wu, and J. Wang, "PLVER: joint stable allocation and content replication for edge-assisted live video delivery," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, under the second round of review

Bibliography

[1] Cisco, "Global mobile data traffic forecast update, 2015–2020," *White Paper, February*, vol. 3, 2016.

[2] 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 Computer Communication Review*, vol. 45, no. 4, pp. 311–324, 2015.

[3] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017, pp. 197–210.

[4] D. Puthal, M. S. Obaidat, P. Nanda, M. Prasad, S. P. Mohanty, and A. Y. Zomaya, "Secure and sustainable load balancing of edge data centers in fog computing," *IEEE Communications Magazine*, vol. 56, no. 5, pp. 60–65, 2018.

[5] J. Yao, H. Zhou, J. Luo, X. Liu, and H. Guan, "COMIC: Cost optimization for internet content multihoming," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 26, no. 7, pp. 1851–1860, 2015.

[6] H. H. Liu, Y. Wang, Y. R. Yang, H. Wang, and C. Tian, "Optimizing cost and performance for content multihoming," in *Proceedings of SIGCOMM*. ACM, 2012, pp. 371–382.

[7] M. K. Mukerjee, I. N. Bozkurt, D. Ray, B. M. Maggs, S. Seshan, and H. Zhang, "Redesigning cdn-broker interactions for improved content delivery," in *Proceedings of International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2017.

[8] X. Liu, F. Dobrian, H. Milner, J. Jiang, V. Sekar, I. Stoica, and H. Zhang, "A case for a coordinated internet video control plane," in *Proceedings of the*

ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication. ACM, 2012, pp. 359–370.

[9] R. Torres, A. Finamore, J. R. Kim, M. Mellia, M. M. Munafo, and S. Rao, "Dissecting video server selection strategies in the youtube cdn," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on.* IEEE, 2011, pp. 248–257.

[10] P. Dogga, S. Chakraborty, S. Mitra, and R. Netravali, "Edge-based transcoding for adaptive live video streaming," in *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[11] "Under the hood: Broadcasting live video to millions," `https://code.fb.com/ios/under-the-hood-broadcasting-live-video-to-millions/`, accessed in April 2020.

[12] "How Facebook Live Streams To 800,000 Simultaneous Viewers," `http://highscalability.com/blog/2016/6/27/how-facebook-live-streams-to-800000-`
accessed in April 2020.

[13] B. Wang, X. Zhang, G. Wang, H. Zheng, and B. Y. Zhao, "Anatomy of a personalized livestreaming system," in *Proceedings of the ACM Internet Measurement Conference*, 2016, pp. 485–498.

[14] G. Yi, D. Yang, A. Bentaleb, W. Li, Y. Li, K. Zheng, J. Liu, W. T. Ooi, and Y. Cui, "The acm multimedia 2019 live video streaming grand challenge," in *Proceedings of the 27th ACM International Conference on Multimedia*, 2019, pp. 2622–2626.

[15] H. Pang, C. Zhang, F. Wang, H. Hu, Z. Wang, J. Liu, and L. Sun, "Optimizing personalized interaction experience in crowd-interactive livecast: A cloud-edge approach," in *Proceedings of the 26th ACM International Conference on Multimedia*, 2018, pp. 1217–1225.

[16] M. Ma, Z. Wang, K. Yi, J. Liu, and L. Sun, "Joint request balancing and content aggregation in crowdsourced cdn," in *Proceedings of Int'l Conf. on Distributed Computing Systems (ICDCS).* IEEE, 2017, pp. 1178–1188.

[17] B. Rainer, D. Posch, and H. Hellwagner, "Investigating the performance of pull-based dynamic adaptive streaming in ndn," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 34, no. 8, pp. 2130–2140, 2016.

[18] C. Ge, N. Wang, W. K. Chai, and H. Hellwagner, "Qoe-assured 4k http live streaming via transient segment holding at mobile edge," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 36, no. 8, pp. 1816–1830, 2018.

[19] A. Biliris, C. Cranor, F. Douglis, M. Rabinovich, S. Sibal, O. Spatscheck, and W. Sturm, "Cdn brokering," *Computer Communications*, vol. 25, no. 4, pp. 393–402, 2002.

[20] "IETF: Content Distribution Internetworking (cdi)," `https://datatracker.ietf.org/wg/cdi/documents/`, 2003, [accessed Dec. 2017].

[21] M. Latouche, J. Defour, T. Renger, T. Verspeht, and F. Lefaucheur, "Cisco Report: The CDN Federation-Solutions for SPs and Content Providers To Scale a Great Customer Experience," 2012, [accessed Dec. 2017].

[22] L. Peterson, B. Davie, and R. van Brandenburg, "RFC 7336: Framework for Content Distribution Network Interconnection (CDNI)," `https://www.rfc-editor.org/info/rfc7336`, 2014.

[23] B. Niven-Jenkins, R. Murray, M. Caulfield, and K. Ma, "RFC 8006: Content Delivery Network Interconnection (CDNI) Metadata," `https://www.rfc-editor.org/info/rfc8006`, 2016.

[24] R. Murray and B. Niven-Jenkins, "RFC 8007: Content Delivery Network Interconnection (CDNI) Control Interface/Triggers," `https://www.rfc-editor.org/info/rfc8007`, 2016.

[25] H. Wang, G. Tang, K. Wu, and J. Fan, "Speeding up multi-cdn content delivery via traffic demand reshaping," in *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 422–433.

[26] T. Stockhammer, "Dynamic adaptive streaming over http–: standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*. ACM, 2011, pp. 133–144.

[27] I. Sodagar, "The mpeg-dash standard for multimedia streaming over the internet," *IEEE MultiMedia*, vol. 18, no. 4, pp. 62–67, 2011.

[28] J. Roger Pantos, William May, "RFC 8216: HTTP Live Streaming," `https://tools.ietf.org/html/rfc8216`.

[29] X. Nie, Y. Zhao, D. Pei, G. Chen, K. Sui, and J. Zhang, "Reducing web latency through dynamically setting tcp initial window with reinforcement learning," in *Quality of Service (IWQoS), 2018 IEEE/ACM 26th International Symposium on*. IEEE, 2018.

[30] J. Jiang, S. Sun, V. Sekar, and H. Zhang, "Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 393–406.

[31] T. Mauro, "Why Netflix Chose NGINX as the Heart of Its CDN," `https://www.nginx.com/blog/why-netflix-chose-nginx-as-the-heart-of-its-cdn/`.

[32] H. Wang, K. Wu, J. Wang, and G. Tang, "Rldish: Edge-assisted qoe optimization of http live streaming with reinforcement learning," in *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2020, pp. 706–715.

[33] M. Ma, Z. Wang, K. Yi, J. Liu, and L. Sun, "Joint Request Balancing and Content Aggregation in Crowdsourced CDN," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017.

[34] V. K. Adhikari, Y. Guo, F. Hao, V. Hilt, Z.-L. Zhang, M. Varvello, and M. Steiner, "Measurement study of Netflix, Hulu, and a tale of three CDNs," *IEEE/ACM Transactions on Networking (TON)*, 2015.

[35] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia, "Reducing latency via redundant requests: Exact analysis," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 347–360, 2015.

[36] "jet-stream," `http://www.jet-stream.com/`, 2017, [accessed Dec. 2017].

[37] M. Day, B. Cain, G. Tomlinson, P. Rzewski, "RFC 3466: A Model for Content Internetworking (CDI)," https://datatracker.ietf.org/doc/rfc3466/, 2003.

[38] "IETF: Content Delivery Networks Interconnection (cdni)," https://datatracker.ietf.org/wg/cdni/documents/, 2017, [accessed Dec. 2017].

[39] G. P. Zhang, "Time series forecasting using a hybrid ARIMA and neural network model," *Neurocomputing*, vol. 50, pp. 159–175, 2003.

[40] G. Tang, K. Wu, and R. Brunner, "Rethinking cdn design with distributed time-varying traffic demands," in *Proceedings of International Conference on Computer Communications (INFOCOM)*. IEEE, 2017.

[41] E. Gourdin, P. Maillé, G. Simon, and B. Tuffin, "The Economics of CDNs and Their Impact on Service Fairness," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 14, no. 1, pp. 22–33, 2017.

[42] V. Valancius, C. Lumezanu, N. Feamster, R. Johari, and V. V. Vazirani, "How many tiers?: pricing in the internet transit market," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 194–205.

[43] Gurobi Optimization, http://www.gurobi.com/.

[44] H. Yin, X. Zhang, S. Zhao, Y. Luo, C. Tian, and V. Sekar, "Trade-offs between Cost and Performance for CDN Provisioning Based on Coordinate Transformation," *IEEE Transactions on Multimedia (TMM)*, 2017.

[45] B. Frank, I. Poese, Y. Lin, G. Smaragdakis, A. Feldmann, B. Maggs, J. Rake, S. Uhlig, and R. Weber, "Pushing CDN-ISP collaboration to the limit," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 3, pp. 34–44, 2013.

[46] F. Le Faucheur, G. Bertrand, I. Oprescu, R. Peterkofsky, "RFC 7937: Content Distribution Network Interconnection (CDNI) Logging Interface," https://datatracker.ietf.org/doc/rfc7937, 2016.

[47] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 29, no. 9, pp. 1765–1775, 2011.

[48] O. Krajsa and L. Fojtova, "Rtt measurement and its dependence on the real geographical distance," in *Telecommunications and Signal Processing (TSP), 2011 34th International Conference on*. IEEE, 2011, pp. 231–234.

[49] L. Peterson, B. Davie, and R. van Brandenburg, "RFC 6707: Content Distribution Network Interconnection (CDNI) Problem Statement," `https://www.rfc-editor.org/info/rfc6707`, 2012.

[50] C. Ge, N. Wang, G. Foster, and M. Wilson, "Toward qoe-assured 4k video-on-demand delivery through mobile edge virtualization with adaptive prefetching," *IEEE Transactions on Multimedia (TMM)*, vol. 19, no. 10, pp. 2222–2237, 2017.

[51] A. Detti, B. Ricci, and N. Blefari-Melazzi, "Tracker-assisted rate adaptation for mpeg dash live streaming," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2016, pp. 1–9.

[52] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 325–338.

[53] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli, "Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 272–285.

[54] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," *Communications of the ACM*, vol. 56, no. 3, pp. 91–99, 2013.

[55] S. S. Krishnan and R. K. Sitaraman, "Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs," *IEEE/ACM Transactions on Networking (TON)*, vol. 21, no. 6, pp. 2001–2014, 2013.

[56] J. Bruneau-Queyreix, M. Lacaud, and D. Négru, "A hybrid p2p/multi-server quality-adaptive live-streaming solution enhancing end-user's qoe," in *Proceedings of the 25th ACM International Conference on Multimedia*. ACM, 2017, pp. 1261–1262.

[57] F. Wang, C. Zhang, J. Liu, Y. Zhu, H. Pang, L. Sun *et al.*, "Intelligent edge-assisted crowdcast with deep reinforcement learning for personalized qoe," in *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE, 2019, pp. 910–918.

[58] K. Miller, A.-K. Al-Tamimi, and A. Wolisz, "Qoe-based low-delay live streaming using throughput predictions," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, vol. 13, no. 1, p. 4, 2017.

[59] T. C. Thang, H. T. Le, A. T. Pham, and Y. M. Ro, "An evaluation of bitrate adaptation methods for http live streaming," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 32, no. 4, pp. 693–705, 2014.

[60] R. Huysegems, J. Van Der Hooft, T. Bostoen, P. Rondao Alface, S. Petrangeli, T. Wauters, and F. De Turck, "Http/2-based methods to improve the live experience of adaptive streaming," in *Proceedings of the 23rd ACM International Conference on Multimedia*. ACM, 2015, pp. 541–550.

[61] A. Garivier and E. Moulines, "On upper-confidence bound policies for switching bandit problems," in *International Conference on Algorithmic Learning Theory (ALT)*. Springer, 2011, pp. 174–188.

[62] A. Ahmed, Z. Shafiq, and A. Khakpour, "Qoe analysis of a large-scale live video streaming event," *ACM SIGMETRICS Performance Evaluation Review*, vol. 44, no. 1, pp. 395–396, 2016.

[63] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, "A survey on quality of experience of http adaptive streaming," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 469–492, 2014.

[64] "hls.js," `https://github.com/video-dev/hls.js/`.

[65] "Docker," `https://www.docker.com/`.

[66] "FFmpeg," `https://ffmpeg.org/`.

[67] S. Tavakoli, S. Egger, M. Seufert, R. Schatz, K. Brunnström, and N. Garcia, "Perceptual quality of http adaptive streaming strategies: Cross-experimental analysis of multi-laboratory and crowdsourced subjective studies," *IEEE Journal*

on Selected Areas in Communications (JSAC), vol. 34, no. 8, pp. 2141–2153, 2016.

[68] "Twitch Statistics and Charts:Twitch Tracker," `https://twitchtracker.com/statistics`, accessed in April 2020.

[69] D. Ray, J. Kosaian, K. Rashmi, and S. Seshan, "Vantage: optimizing video upload for time-shifted viewing of social live streams," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 380–393.

[70] H. Xu and B. Li, "Joint request mapping and response routing for geo-distributed cloud services," in *Proceedings of Int'l Conf. on Computer Communications (INFOCOM)*. IEEE, 2013, pp. 854–862.

[71] S. Narayana, J. W. Jiang, J. Rexford, and M. Chiang, "To coordinate or not to coordinate? wide-area traffic management for data centers," *Dept. Comput. Sci., Princeton Univ., Princeton, NJ, USA, Tech. Rep. TR-998-15*, 2012.

[72] B. Yan, S. Shi, Y. Liu, W. Yuan, H. He, R. Jana, Y. Xu, and H. J. Chao, "Livejack: Integrating cdns and edge clouds for live content broadcasting," in *Proceedings of the 25th ACM International Conference on Multimedia*, 2017, pp. 73–81.

[73] Y. Zhang, C. Gao, Y. Guo, K. Bian, X. Jin, Z. Yang, L. Song, J. Cheng, H. Tuo, and X. Li, "Proactive video push for optimizing bandwidth consumption in hybrid cdn-p2p vod systems," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 2555–2563.

[74] V. Joseph and G. de Veciana, "Nova: Qoe-driven optimization of dash-based video delivery in networks," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 82–90.

[75] J. Kim, G. Caire, and A. F. Molisch, "Quality-aware streaming and scheduling for device-to-device video delivery," *IEEE/ACM Transactions on Networking*, vol. 24, no. 4, pp. 2319–2331, 2016.

[76] Z. Lu and G. De Veciana, "Optimizing stored video delivery for mobile networks: The value of knowing the future," *IEEE Transactions on Multimedia*, 2018.

[77] H. Hu, Y. Wen, T.-S. Chua, J. Huang, W. Zhu, and X. Li, "Joint content replication and request routing for social video distribution over cloud cdn: A community clustering method," *IEEE transactions on circuits and systems for video technology*, vol. 26, no. 7, pp. 1320–1333, 2016.

[78] Y. Zhou, L. Chen, C. Yang, D. M. Chiu *et al.*, "Video popularity dynamics and its implication for replication." *IEEE Trans. Multimedia*, vol. 17, no. 8, pp. 1273–1285, 2015.

[79] D. Applegate, A. Archer, V. Gopalakrishnan, S. Lee, and K. K. Ramakrishnan, "Optimal content placement for a large-scale vod system," *IEEE/ACM Transactions on Networking*, vol. 24, no. 4, pp. 2114–2127, 2015.

[80] A. O. Al-Abbasi and V. Aggarwal, "Edgecache: An optimized algorithm for cdn-based over-the-top video streaming services," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018, pp. 202–207.

[81] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 3, pp. 52–66, 2015.

[82] D. Gusfield and R. W. Irving, *The stable marriage problem: structure and algorithms*. MIT press, 1989.

[83] C. Chekuri and S. Khanna, "A polynomial time approximation scheme for the multiple knapsack problem," *SIAM Journal on Computing*, vol. 35, no. 3, pp. 713–728, 2005.

[84] S. Laabadi, M. Naimi, H. El Amri, B. Achchab *et al.*, "The 0/1 multidimensional knapsack problem and its variants: a survey of practical models and heuristic approaches," *American Journal of Operations Research*, vol. 8, no. 05, p. 395, 2018.

[85] A. Fréville, "The multidimensional 0–1 knapsack problem: An overview," *European Journal of Operational Research*, vol. 155, no. 1, pp. 1–21, 2004.

[86] "Twitch dataset," `http://dash.ipv6.enstb.fr/dataset/live-sessions/`.

[87] K. Pires and G. Simon, "Youtube live and twitch: a tour of user-generated live streaming systems," in *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 2015, pp. 225–230.

[88] "Youtube. Live encoder settings, bitrates, and resolutions," `https://support.google.com/youtube/answer/2853702?hl=en`, accessed in May 2019.

[89] "geoisp," `https://geoisp.com/us/`, accessed in May 2019.

[90] Y.-H. Hung, C.-Y. Wang, and R.-H. Hwang, "Combinatorial clock auction for live video streaming in mobile edge computing," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2018.

[91] H. Wang, G. Tang, K. Wu, and J. Wang, "PLVER: joint stable allocation and content replication for edge-assisted live video delivery," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, under the second round of review.