TailCutter: Wisely Cutting Tail Latency in Cloud CDNs Under Cost Constraints

Yong Cui, Ningwei Dai, Zeqi Lai, Minming Li, Zhenhua Li, *Member, IEEE*, Yuming Hu, Kui Ren, *Fellow, IEEE, Member, ACM*, and Yuchi Chen

***Abstract*— Cloud computing platforms enable applications to offer low-latency services to users by deploying data storage in multiple geo-distributed data centers. In this paper, through benchmark measurements on Amazon AWS and Microsoft Azure together with an analysis of a large-scale dataset collected from a major cloud CDN provider, we identify the *high tail latency problem* in cloud CDNs, which can substantially undermine the efficacy of cloud CDNs. One crucial idea to reduce the tail latency is to send requests in parallel to multiple clouds in cloud CDNs. However, since application providers often have a budget for using cloud services, deciding how many chunks to download from each cloud and when to download chunks in a cost-efficient manner still remain as open problems in our concerned scenario. To address the problem, we present TailCutter, a workload scheduling framework that aims at optimizing the tail latency while meeting cost constraints given by application providers. Specifically, we formulate the *tail latency minimization* (TLM) problem in cloud CDNs and design the *receding horizon control based maximum tail minimization algorithm* (RHC-based MTMA) to efficiently solve the TLM problem in practice. We implement TailCutter across multiple data centers of Amazon AWS and Microsoft Azure. Extensive evaluations using a large-scale real-world data trace (collected from a major ISP) illustrate that TailCutter can reduce up to 58.9% of the 100th-percentile user-perceived latency, as compared with alternative solutions under the cost constraint.**

***Index Terms*— Cloud storage, tail latency, measurement, optimization.**

I. Introduction

Cloud storage services are gaining tremendous popularity in recent years by providing the appealing benefits of low maintenance, easy access and elasticity for geo-distributed online data storage. A recent study shows that the global cloud storage market is expected to reach \$56.57 billion by 2019, with a compound annual growth rate of 33.1% [1]. The recent prosperities of cloud storage providers such as Amazon AWS and Microsoft Azure are notable examples. Cloud storage providers operate data centers that can offer Internet-based content storage and delivery capabilities with the assurance of service uptime and end-user perceived service quality.

The emergence of cloud storage services provides new opportunities for application providers. In particular, application providers can build a cloud storage-based content delivery network (abbreviated as a "*cloud CDN*") to provide low-latency services to users without the high cost and complexity of owning and operating geographically dispersed data centers by themselves. As the user access latency is one of the most important QoS metrics, a large amount of previous efforts [2]–[13] focused on how to place data replicas in different clouds and optimize the latency performance in cloud CDNs.

In many distributed systems, fetching data from a single serving node is often associated with *high latency variance*. This phenomenon, often called the "tail latency", exists not only in modern dedicated data centers [14], [15], but also in cloud CDNs. The impact of high latency variance is problematic for popular applications where even 1% of traffic corresponds to a significant volume of user requests [16], and for applications where the user is required to download several objects and the user-perceived latency is constrained by the object downloaded last (e.g. downloading chunks of a large file distributedly stored in the Dropbox server).

As the first contribution in this paper, we identify the *high user-perceived tail latency problem* in cloud CDNs, which may significantly degrade user experience but has not yet been efficiently addressed (§II). Our benchmark measurements on Amazon AWS and Microsoft Azure show that the high latency variance exists both within the cloud data center and over the Internet. Quantitatively, we further analyze a recent large-scale dataset collected from a major cloud CDN provider named Xuanfeng [17], [18]. The dataset records the latency of 4,084,417 file downloads in a whole week, involving 783,944 users and 563,517 unique files. Surprisingly, we find that *the 100th-percentile download latency can be up to 1076$\times$ of the median!* These variances are caused by many factors, including the competition of shared resources, failure of equipment, link congestions, etc. [19], and it seems almost impossible to avoid such variances. Therefore a feasible latency optimization should live with the high variance.

To address the high tail latency problem in cloud CDNs and enable application providers to avail of the low-cost benefits provided by cloud services, we propose TailCutter, a novel workload scheduling framework lying between users

Manuscript received August 25, 2016; revised May 22, 2017, February 22, 2018 and April 24, 2019; accepted June 15, 2019; approved by IEEE/ACM Transactions on Networking Editor M. Chen. Date of publication August 6, 2019; date of current version August 16, 2019. This work was supported in part by the NSFC under Grant 61872211, and in part by the National Key R&D Program of China under Grant 2017YFB1010002. *(Corresponding author: Zeqi Lai.)*

Y. Cui, N. Dai, Z. Lai, and, Y. Hu are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: cuiyong@tsinghua.edu.cn; uestclzq@gmail.com; lemondnw@gmail.com; yuming.hum@gmail.com.

M. Li is with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: minming.li@cityu.edu.hk).

Z. Li is with the School of Software, Tsinghua University, Beijing 100084, China (e-mail: lizhenhua1983@gmail.com).

K. Ren is with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China (e-mail: kuiren@zju.edu.cn).

Y. Chen is with the School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada (e-mail: chenycmx@gmail.com).

Digital Object Identifier 10.1109/TNET.2019.2926142

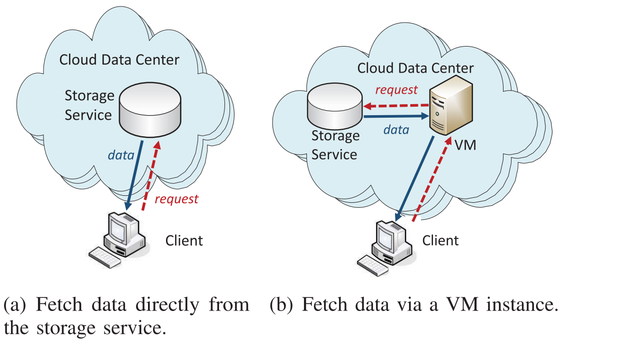
and cloud storage data centers. There are many recent studies in dedicated data centers that optimize tail latency by sending redundant requests to multiple serving nodes and using the response first returned [7], [14], [15], [20], [21]. Inspired by the above studies, the key idea behind TAILCUTTER is to issue requests in parallel to download different chunks of the file from different clouds while meeting cost constraints, under the assumption that files are replicated in each cloud. Highly targeting at the cloud CDN scenario, TAILCUTTER makes several domain-specific optimizations: (1) Since cloud storage providers charge their customers (partially) according to their bandwidth usage, and application providers often have a budget for using cloud storage services, TAILCUTTER optimizes the tail latency under the cost constraint given by application providers; (2) Instead of using redundant requests, TAILCUTTER issues multiple requests to download different parts of the replica located in multiple clouds to avoid bandwidth wastes; (3) To prevent link congestions caused by bandwidth competition or other sources in data centers, TAILCUTTER stands in a central view to schedule all requests, decide when and how to download chunks from each cloud, and manage user bandwidth, cloud bandwidth, and bandwidth between user and cloud.

More specifically, in this paper we first formulate the *Tail Latency Minimization (TLM)* problem that aims at optimizing the tail latency while meeting application providers' cost constraint in cloud CDNs (§III). To our best knowledge, we are the first to optimize the tail latency under the cost constraints in cloud CDNs. Given the formulation, we then design the *Maximum Tail Minimization Algorithm (MTMA)* which solves the TLM problem in polynomial time (§IV). However the applicability of MTMA is limited since it requires the complete knowledge of the available bandwidth of each cloud, the end-to-end bandwidth, and the workload of all requests in an entire scheduling period. To further address this limitation, we extend MTMA and design a more practical online algorithm called *Receding Horizon Control (RHC) based MTMA*, which schedules requests based on the bandwidth prediction in a short future horizon and adjusts the scheduling results over time to efficiently and robustly optimize the tail latency under the cost constraints.

We implement TAILCUTTER system across a set of data centers in Amazon AWS and Microsoft Azure (§V). We conduct a trace-driven evaluation on our prototype using a 2-Terabyte real-world data trace (collected from a major ISP), and the results demonstrate that TAILCUTTER is able to reduce the tail latency by up to 73.3% as compared to other state-of-the-art solutions. In addition, we also conduct a large-scale simulation to evaluate the scalability of TAILCUTTER. The results of the trace-driven simulation show that TAILCUTTER scales well under different sizes of workloads, and can effectively cut up to 58.9% of the 100th-percentile tail latency without draining the budget of application providers.

II. MOTIVATION

We begin with a measurement study to motivate our work. In this section, we first quantify the *high tail latency* problem in cloud CDNs. Then we introduce and analyze the opportunity that leverages multiple clouds to reduce tail latency. Finally we highlight the importance of considering cost constraint when optimizing tail latency in cloud CDNs.



(a) Fetch data directly from the storage service. (b) Fetch data via a VM instance.

Fig. 1. Illustration of the typical approaches to build cloud CDNs upon cloud data centers.

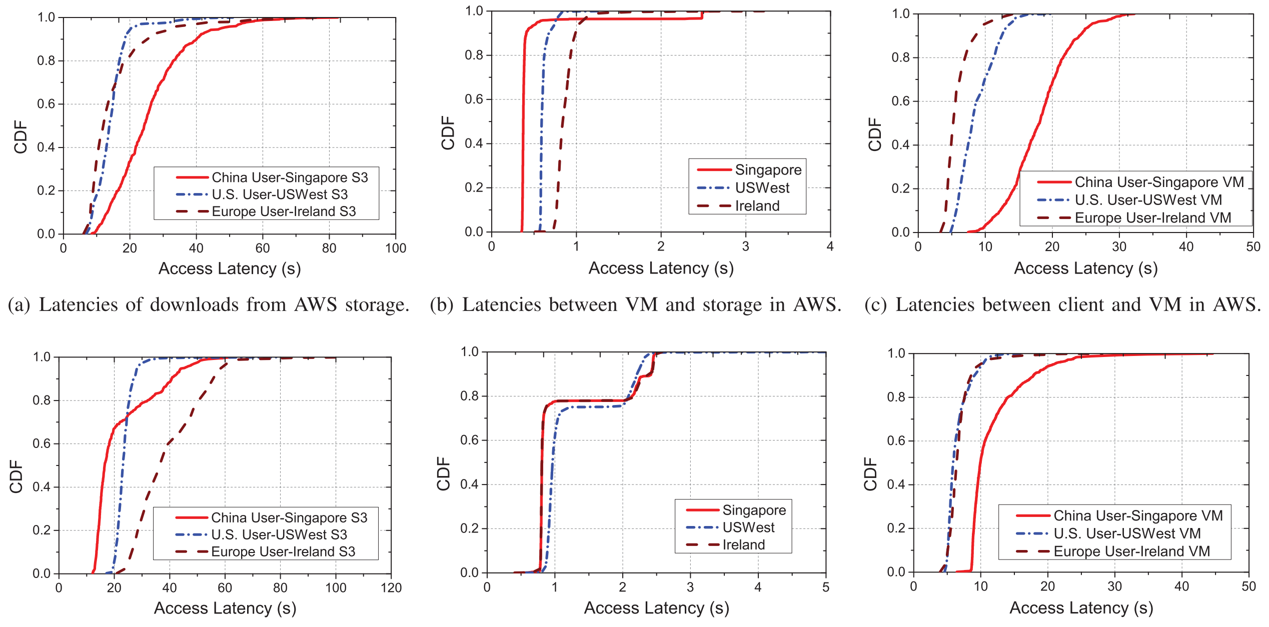
A. Background of Cloud CDNs

Cloud CDNs provide low-latency services to users without the high cost and complexity of owning and operating geographically dispersed data centers. Figure 1 illustrates the typical approach of building cloud CDNs upon cloud storage services. The application provider often stores data objects in the cloud rented from cloud providers (e.g. Amazon AWS or Microsoft Azure) to build the cloud CDN. In particular, the client application can directly fetch data from the storage service inside a cloud data center (e.g. Amazon S3) as shown in Figure 1(a), or fetch data relying on an opened Virtual Machine (VM) in the same data center (e.g. Amazon EC2) as shown in Figure 1(b). The latter pattern is usually used in scenarios with packet processing, e.g. the VM is used for request authentication or data encryption/decryption.

B. Quantifying the High Tail Latency in Cloud CDNs

To quantify the high tail latency problem, we conduct a measurement study upon two popular cloud providers Amazon AWS and Microsoft Azure, together with an in-depth analysis on a large-scale data set collected from a commercial cloud CDN. Quantitatively, we define the *user-perceived access latency* as the time for a user to completely download a certain file from a cloud. In cloud CDNs, high latency variance is very problematic because for popular applications even only 1% of their traffic corresponds to a significant volume of requests [16]. In addition, worst-case performance matters much more to applications that fetch multiple objects and the operation completion time is constrained by the object fetched last (e.g. downloading a large file in Dropbox which is split into chunks and distributedly stored on the server). In the following of this paper we use 95th, 99th, and 100th percentile latency of all users to quantify the tail latency in practice.

Quantifying Tail Latencies on AWS and Azure: To quantify the access latency we develop a measurement tool that contains a client and a server counterpart running on the cloud data center. The client periodically sends requests to the server to download files directly from the storage service or via a



(a) Latencies of downloads from AWS storage. (b) Latencies between VM and storage in AWS. (c) Latencies between client and VM in AWS.

(d) Latencies of downloads from Azure storage. (e) Latencies between VM and storage in Azure. (f) Latencies between client and VM in Azure.

Fig. 2. High latency variance of downloading data from Amazon AWS or Windows Azure.

VM instance as shown in Figure 1, and then measure the access latency of each request. Particularly, we deploy the client on three machines located at U.S., Europe, and China respectively, and run the server on the nearest Amazon AWS or Microsoft Azure cloud data center located at west U.S., Ireland and Singapore respectively. In each test, we issue a single GET request from the client to download a 10MB file in every 2 mins in one day. We use `tcpdump` to record the packet-level network traces and calculate the user-perceived access latency. More specifically, for tests downloading files directly like Figure 1(a), we measure the latency as the time of completely downloading the file from the storage service. For tests where the transmission relies on a VM instance as shown in Figure 1(b), we measure and break down the total latency into (1) the latency of fetching the file from the storage service to the VM, and (2) the latency of delivering the file from the VM to the client.

The cumulative distribution function (CDF) of the measured access latency is plotted in Figure 2. We have made the following key observations: (1) The access latencies of directly downloading data from the storage service suffer from significant variation. As shown in Figure 2(a) and 2(d), the 99th percentile access latency is at least 2× and up to 5× of the median, and the 100th percentile access latency is at least 2.8× and up to 6.4× of the median; (2) Downloading data via a VM instance also suffers from high latency variation within the data center and over the Internet. The 99th percentile latency is 1.4-3.8× of the median, and the 100th percentile latency is 1.8-6.7× of the median between the VM and storage service in the same cloud as shown in Figure 2(b) and 2(e); The 99th percentile latency is 1.7-3.3× of the median, and the 100th percentile latency is 2.1-7.2× of the median from the VM to the client as plotted in Figure 2(c) and 2(f); (3) For VM-relied downloads, the total access latency is dominated by the latency between the client and the VM in the cloud; (4) High tail latency is prevalent in cloud CDNs since it is observed in both cloud providers and in different locations.

Analyzing Tail Latencies on QQ Xuanfeng: To further understand the tail latency problem in cloud CDNs, we analyzed a large-scale dataset collected from a commercial cloud CDN provider, QQ Xuanfeng [17]. QQ Xuanfeng is a major provider of cloud CDN in China, processing over 30 million users at the moment. The dataset we collected contains the complete logs of QQ Xuanfeng for a week (Feb. 22-28, 2015), involving 4,084,417 downloading tasks, 783,944 users and 563,517 unique files. Figure 5 plots the file size distribution of the dataset collected from QQ Xuanfeng, in which large files dominate (e.g. about 90% files are larger than 10MB).

We analyzed the access latency when fetching files differing in sizes. We extract the downloading time and the fetched file size from the dataset, and plot the CDF of the access latency when fetched file size is around 100KB, 1MB and 100MB respectively. As shown in Figure 3, surprisingly the 99th percentile latencies for different file sizes are at least 31× and up to 49× of the median latencies, and the 100th percentile latencies for different file sizes are at least 60× and up to 376× of the median latencies. We also examine the tail latency of downloading files from different IP prefixes, and we plot the latencies of the top four IP subnetworks that have most user requests in Figure 4. We find that the 99th percentile latencies for the same file size from different areas are at least 17× and up to 359× of the median latencies, and the 100th percentile latencies are at least 32× and up to 1076× of the median latencies! The tail latency is much higher in a large-scale commercial cloud CDN as compared to our benchmark measurement, because these latency variances are random and caused by many complex factors, including congestion, shared resources competition, equipment failure, etc., and are almost

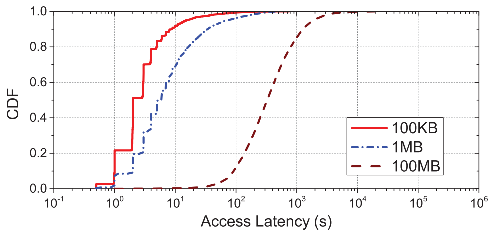


Fig. 3. CDF of access latencies of downloading files differing in sizes.

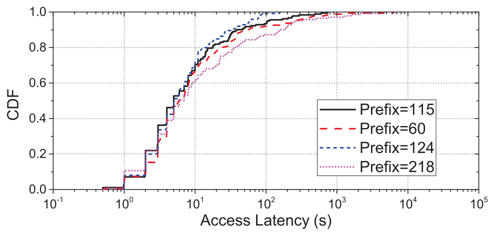


Fig. 4. Access latencies of download requests from different IP prefixes.

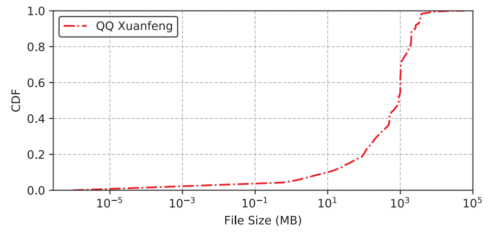


Fig. 5. CDF of file sizes in QQ Xuanfeng traces.

impossible to prevent [3], [19]. Since the above root causes are complex and diverse, in this paper we try to treat tail latency as a black box and propose a comprehensive solution instead of dealing with the sources one by one.

C. Leveraging Multiple Clouds to Reduce Latency

Since the latency of a single cloud data center suffers from significant variance, a straightforward way to reduce tail latency is to issue multiple requests to different data centers to download data. To identify the effectiveness of using multiple clouds, we place the same 10 MB replica on two nearest AWS and Azure data centers. We extend the measurement tool to enable the client to download a file by: (1) sending a single GET request to the nearest data center to get the entire replica; or (2) sending two redundant requests in parallel to the nearest two AWS or two Azure data centers, and the first received file will be used. The second method is originally inspired by previous studies [7], [19], [22] focusing on reducing tail latency in dedicated data centers. However at the same time it incurs additional bandwidth cost due to the redundant requests. In our experiment we propose a third method: (3) the client issues two requests to download different parts of the replica, e.g. one request downloads from the beginning while the other one downloads from the end, and the download process completes once the entire file is collected. Note that in the third method, the data downloaded by each request may differ in size but their sum is equal to the total file size.

We run the extended measurement tool at three locations in U.S., Europe and China, and plot the CDF of the access latency in Figure 6. The experiment provides the following key insights: (1) naively leveraging multiple clouds to download data (e.g. issuing redundant requests to different clouds) reduces the 99th percentile user-perceived latency by at least 11% and up to 61%, and reduces the 100th percentile user-perceived latency by at least 5% and up to 73% on AWS and Azure platforms; (2) the tail latency can be further reduced by avoiding bandwidth wastes and finding a proper data assignment for these requests to complete each of them at the same time. In our experiment, issuing requests to download different parts of replicas significantly reduces 99th percentile user-perceived latency by at least 43% and up to 83%, and reduces 100th percentile user-perceived latency by at least 33% and up to 64%. *The measurement results show that given a proper assignment, leveraging multiple clouds is indeed an effective approach to reduce the latency variability.*

D. Cost-Effectiveness Considerations

To use cloud storage services, application providers pay money to cloud providers according to the bandwidth and the storage usage of their applications. Generally, the bandwidth cost is much higher than the storage cost, and cloud data centers offering better latency/bandwidth performance might be more expensive. Since application providers may have a cost constraint for using cloud storage services, although we can leverage parallel multiple clouds to reduce tail latency in cloud CDNs, the concrete approach for all users to download data (e.g. cloud selection and bandwidth assignment) should be well designed to meet the cost constraint of application providers.

E. Lessons Learned

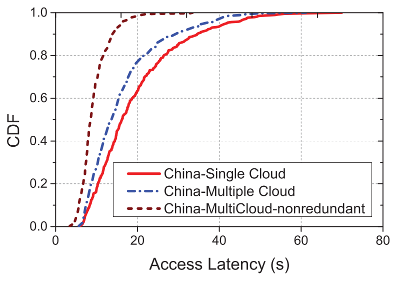
In summary, we have identified three key insights from our measurement study: (1) modern cloud data centers inevitably suffer from high tail latency, which may significantly impair the user experience; (2) a promising opportunity for optimizing tail latency is to properly schedule requests to multiple clouds and manage the bandwidth for each request; (3) when designing such a scheduler, the total cost incurred by data transmission should not exceed the budget given by application providers.

III. TailCutter Overview and Problem Formulation

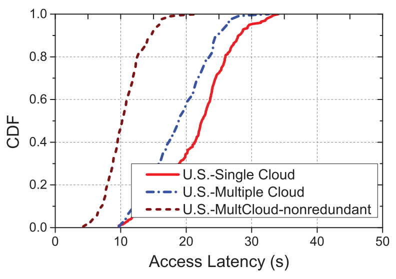
Motivated by the key insights obtained from our measurement study, we propose TailCutter, a novel workload scheduling framework in cloud CDNs to solve the high latency problem. In this section, we introduce the system overview of TailCutter and then formulate the Tail Latency Minimization (TLM) problem that aims at optimizing the user-perceived latency while meeting the constraint of cost overhead.

A. TailCutter *Overview*

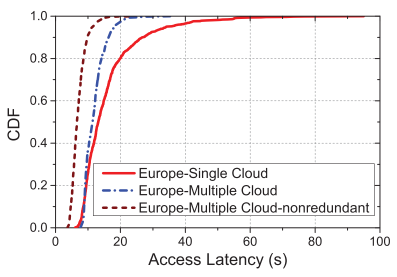
As shown in Figure 7 TailCutter contains three key components in high level: the *Request Scheduler*, the *Measurement*



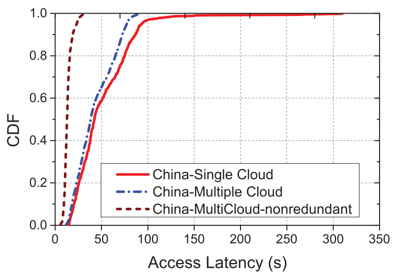
(a) Latency variance, AWS, China



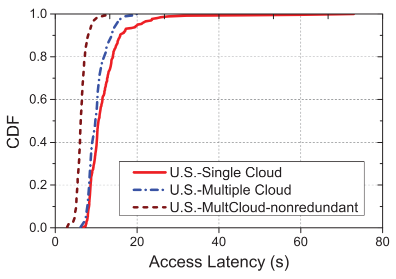
(b) Latency variance, AWS, U.S.



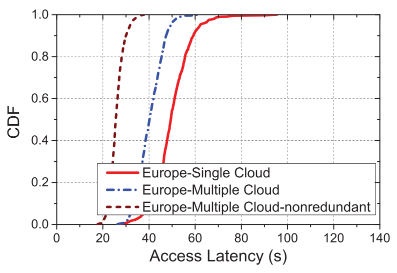
(c) Latency variance, AWS, Europe



(d) Latency variance, Azure, China



(e) Latency variance, Azure, U.S.



(f) Latency variance, Azure, Europe

Fig. 6. Tail latency reduction by leveraging multiple clouds.

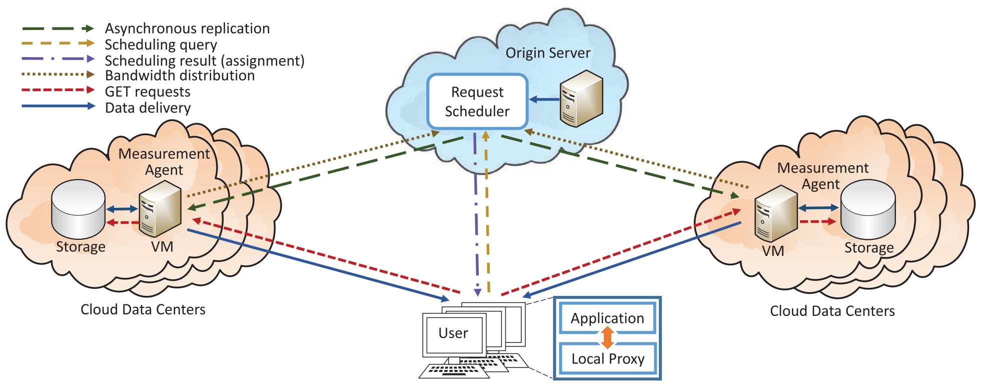


Fig. 7. TAILCUTTER system overview and the problem scenario.

Agent, and the *Local Proxy* that work together to automatically schedule user requests and manage the bandwidth.

Local Proxy: To execute a download task, the application (e.g. a download manager for shared file systems or video games) on end users links to TAILCUTTER's Local Proxy, which first sends a schedule query to the Request Scheduler to obtain an *assignment*, and then follows the assignment to download data from cloud CDNs. Specifically, the assignment tells the user which cloud the client should send requests to, and how much data to download from each cloud. Once the assignment is received, the Local Proxy issues a set of GET requests to different clouds. To reduce the bandwidth overhead, GET requests issued by the Local Proxy download different parts of the replica located in different clouds.

Measurement Agent: In every potential cloud, we build the Measurement Agent to measure the bandwidth knowledge of the cloud (e.g. the bandwidth distribution between the cloud and every end user). The Measurement Agent periodically sends the observed bandwidth knowledge to the central Request Scheduler for further scheduling. Particularly, for a practical storage service in a cloud data center, e.g. Amazon AWS, it is difficult to directly obtain the bandwidth knowledge because of the limitation of the exposed storage APIs. Therefore, we deploy the Measurement Agent in a VM instance as a gateway of the storage service in the same data center. The number of Measurement Agents (gateways) on a cloud can be configured by the application provider. If more Measurement Agents than one are configured, TailCutter can set multiple Measurement Agents and assign each Measurement Agent to measure the network conditions of different storage nodes on the same cloud.

In practice, measuring the end-to-end bandwidth knowledge in TailCutter does not involves significant overhead due to the following reasons: (1) TailCutter provides each user a limited set of cloud servers based on its location (e.g. only the nearest N clouds are available for a certain user), and it is unnecessary for each user to establish a connection to every cloud data center to measure the end-to-end bandwidth; (2) since the network performance might be similar for all users from the same IP prefix to any particular data center [23], TailCutter leverages the historical information (e.g. the bandwidth sampled in last slot) in the same IP prefix to approximate the end-to-end bandwidth; (3) only under certain circumstances where no historical bandwidths can be used (e.g. for a new user), the user has to establish a new connection to each available cloud servers and probe the bandwidth by fetching the first data chunk (similar to the method used in [24]); (4) the bandwidth knowledge of a cloud can be delivered via a small HTTP request, and is sent to the Request Scheduler in every slot which is about tens of seconds so that the communication overhead is inexpensive.

Request Scheduler: As the core component in TAILCUTTER framework, the Request Scheduler maintains a single queue for all requests, and leverages the workload and bandwidth knowledge in each cloud to schedule user requests from all users to cut tail latency. The application provider using the TAILCUTTER framework owns an origin server that stores all original data, which are linked to the Request Scheduler located at the origin server. Replicas of the original data are delivered to different cloud data centers asynchronously to accelerate the download performance of geo-distributed users. TAILCUTTER's central Request Scheduler periodically collects the bandwidth knowledge of cloud data centers from each Measurement Agent, and determines the concrete assignment for each user that specifies: (1) when to issue chunk requests, (2) which cloud the client should issue chunk requests to, and (3) how many chunks should be downloaded from each cloud. Collectively, the decision is made according to the workload and bandwidth distribution observed in each cloud data center, the bandwidth distribution between each end user and the cloud, bandwidth purchased by each user, the pricing policy of different cloud providers, as well as the cost overhead constraints given by the application provider. We will revisit more details about how each component works together to accomplish the optimization for tail latency in Section IV-C.

B. Problem Formulation

Now we formulate the *Tail Latency Minimization (TLM)* problem across multiple clouds. In reality, a cloud provider owns and operates multiple data centers, and each data center belongs to only one provider. We assume that there are M cloud data centers $C = \{C_1, C_2, ..., C_M\}$ indexed by i, and N end users $U = \{U_1, U_2, ..., U_N\}$ indexed by j. Without loss of generality, if a user requests multiple files, the requests need to be scheduled one by one in our model. For ease of exposition, we assume that time is slotted and each slot lasts for τ seconds. There are K slots in a scheduling period and slots are indexed by k. Requests of different users may arrive at different time slots, and we define A_j as the arrive time slot of U_j. As we have shown in our measurement study (Section II), the access latency of each cloud data center inevitably suffers from high variance because the available bandwidth may change over time. In addition, for application providers who purchase services (e.g. VM instance or storage) on the cloud, the available bandwidth of a cloud site is allocated according to the corresponding price policy [25]. Therefore, let u_{ik} denote the total available output bandwidth of C_i in slot k which is allocated based on the price policy. We denote the bandwidth limit between C_i and U_j in slot k as d_{ijk}. Note that the end-to-end bandwidth is not only limited by the congestion over some intermediate links but also limited by the ISP based on the amount of bandwidth purchased. Allowing users to make multiple requests in parallel can in effect increase their bandwidth, assuming that the bandwidth bottleneck is the data center. If the bottleneck is the last mile, then making multiple requests in parallel will not help. Since the bandwidth a user has purchased is usually a fixed value provided by the ISP, we assume there is an ISP-specific bandwidth constraint b_j for U_j.

Request and Bandwidth Assignment: Assume that for U_j the size of the replica is W_j MB. A replica is split into multiple ω MB small data chunks, and a GET request downloads a certain number of data chunks. Hence we define the *request and bandwidth assignment* vector x_{ijk} as the number of chunks that U_j downloads from C_i in slot k. According to the definitions we derive following properties: (1) the total bandwidth assigned to U_j in slot k is $\omega \cdot \sum_{i=1}^{M} x_{ijk}$, which should not exceed the ISP-specific bandwidth constraint $b_j \cdot \tau$; (2) the total data size received by U_j is calculated as $\omega \cdot \sum_{i=1}^{M} \sum_{k=A_j}^{K} x_{ijk}$, which should be equal to the replica size W_j; (3) the data size delivered by C_i in slot k is $\omega \cdot \sum_{j=1}^{N} x_{ijk}$ and should not exceed $u_{ik} \cdot \tau$; (4) the size of data transferred from C_i to U_j in slot k is $\omega \cdot x_{ijk}$ and should be less than $d_{ijk} \cdot \tau$.

Total Cost: Cloud providers charge for their outgoing traffic. In practice, some cloud providers charge the bandwidth cost according to both the number of GET requests and the amount of delivered traffic size. Moreover, the provider may charge less per unit of data if the total data size grows. For example, AWS charges \$0.155 per GB data for the first 10TB data transferred and it charges \$0.115 per GB data for the next 40TB [26]. In our model, we assume that each download operation from a user to a cloud data center is operated in a single GET request, and we assume cloud C_i charges unit price of G_i per data chunk in a certain scheduling period. G_i may change according to the transferred data size and the specific pricing model of different cloud providers. Therefore, the total cost overhead in a scheduling period is $\sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{k=A_j}^{K} G_i \cdot x_{ijk}$. In our TLM problem, we focus on the bandwidth cost because it is dominant in the overall cost of building a cloud CDN. We denote f as the total cost constraint in a scheduling period.

Fair User-Perceived Latency: Let binary variable y_{jk} denote the transmission state of U_j in slot k. The request of U_j arrives at slot A_j, and y_{jk} is 1 if only U_j has not finished the download process in time slot k after time slot A_j. Note that there might be some time slots in which the transmission is paused and suspended (i.e. no data chunks are transferred to U_j, $\sum_{i=1}^{M} x_{ijk} = 0, k \geq A_j$) and y_{jk} remains at 1 in these paused slots. Therefore y_{jk} is a non-increasing value after the request arrive time A_j. The user-perceived latency of U_j, which is the time from request arrival at A_j to request download completion of U_j, can be formulated as $L_j = \sum_{k=A_j}^{K} y_{jk}$. Note that the size of replica (W_j) differs for different users, thus to attain fairness we propose a fairness factor λ. We calculate the fair user-perceived latency for U_j as $D_j = \frac{L_j}{(W_j/\omega)^\lambda}$. When $\lambda = 0$, D_j is equal to the user-perceived latency of U_j; when $\lambda = 1$, D_j represents the per-chunk latency of U_j. Here λ is a flexible parameter which can be adjusted to attain fairness according to different scenarios. In Section V-A we have a micro benchmark to discuss how to choose the value of λ. Accordingly the maximum fair user-perceived tail latency among all users is calculated as $\mathbf{max}\{D_j\}$.

Objective: Summarily, the primary objective of the TLM problem is to find a proper request and bandwidth assignment for all users that minimizes the fair user-perceived tail latency while satisfying the total cost constraint. Thus the TLM problem can be formulated as follows.

$$\mathbf{min}\{\mathbf{max}\{D_j\}\}, \ \mathbf{where}\ D_j = \frac{L_j}{(W_j/\omega)^\lambda}, \ L_j = \sum_{k=A_j}^{K} y_{jk} \tag{1}$$

subject to:

$$\omega \cdot \sum_{i=1}^{M} \sum_{k=A_j}^{K} x_{ijk} = W_j, \quad \forall j \in U \tag{2}$$

$$\sum_{i=1}^{M} \sum_{j=1}^{N} \sum_{k=A_j}^{K} G_i \cdot x_{ijk} \leq f \tag{3}$$

$$\omega \cdot \sum_{j=1}^{N} x_{ijk} \leq u_{ik} \cdot \tau, \quad \forall i \in C, \ k \in [1, K] \tag{4}$$

$$\omega \cdot x_{ijk} \leq d_{ijk} \cdot \tau, \quad \forall i \in C, \ j \in U, \ k \in [A_j, K] \tag{5}$$

$$\omega \cdot \sum_{i=1}^{M} x_{ijk} \leq b_j \cdot \tau, \quad \forall j \in U, \ k \in [A_j, K] \tag{6}$$

$$\sum_{i=1}^{M} x_{ijk} \leq (W_j/\omega) \cdot y_{jk}, \quad \forall j \in U, \ k \in [A_j, K] \tag{7}$$

$$\sum_{i=1}^{M} x_{ijk} > y_{jk} - y_{j(k+1)} - 1, \quad \forall j \in U, \ k \in [A_j, K] \tag{8}$$

$$y_{jk} \geq y_{j(k+1)}, y_{jk} \in \{0, 1\}, \quad \forall j \in U, \ k \in [A_j, K] \tag{9}$$

$$x_{ijk} \in \mathbb{N}, \quad \forall i \in C, \ j \in U, \ k \in [A_j, K] \tag{10}$$

$$y_{jk} = 0, x_{ijk} = 0, \quad \forall j \in U, \ k \in [1, A_j) \tag{11}$$

Constraint (2) guarantees that the request from U_j downloads W_j data. Constraint (3) indicates that the total bandwidth cost should not exceed the cost limit f in a scheduling period. Constraint (4) requires that in every time slot, the total number of data chunks delivered by C_i should not exceed the cloud capacity. Constraint (5) requires that the available bandwidth between a user and a cloud is limited by the user bandwidth constraint. Constraint (6) requires that the total number of data chunks requested by U_j should not exceed the ISP-specific user bandwidth constraint. The specially designed constraints (7) and (8) indicate that a user can download data chunks from cloud data centers if and only if the transmission does not complete. If k is the last slot of the data transmission for U_j (i.e. $y_{jk} = 1$ and $y_{j(k+1)} = 0$), there must be some data chunks transferred to U_j in k, i.e. $\sum_{i=1}^{M} x_{ijk} > 0$. Constraint (9), (10) and (11) guarantee binary value y_{jk} is non-increasing after time slot A_j, and x_{ijk} is natural number. The notations used in this paper are summarized in Table I.

TABLE I
SUMMARY OF NOTATIONS

Term	Definition
C_i	Cloud data center i
U_j	User j
G_i	Cost per data chunk of C_i (unit: \$)
L_j	User-perceived latency of U_j
D_j	Fair user-perceived latency of U_j
D	The maximum fair user-perceived latency among all users
x_{ijk}	The number of data chunks downloaded by of U_j from C_i in slot k
y_{jk}	The binary metric indicating the transmission state for U_j in slot k
λ	Fairness factor to adjust the fairness of end users
τ	Time of a slot (unit: second)
u_{ik}	Bandwidth of C_i in slot k (unit: MB/s)
d_{ijk}	End-to-end bandwidth between cloud data center i and user j in slot k (unit: MB/s)
b_j	Bandwidth constraint of U_j (unit: MB/s)
ω	The data size of each data chunk
W_j	Total data size requested by U_j (unit: MB)
A_j	The arrive time of the request from U_j
f	The total cost constraint in a scheduling period

Our TLM problem is essentially an instance of the Integer Linear Programming (ILP) problem. We implement the ILP formulation and solve it with CPLEX [27]. The typical running time for the solver ranges from minutes to hours which is too long to be practical. This also elicits the need for a more efficient solution.

IV. SCHEDULING ALGORITHM IN TAILCUTTER

Now we introduce the design details of the scheduler in TAILCUTTER framework. We first study a simplified scenario with only two clouds in a single slot, and then propose the *Maximum Tail Minimization Algorithm* (MTMA) to solve the TLM problem. In addition, we propose the online *Receding Horizon Control based (RHC-based) MTMA* to schedule user requests more effectively in practical scenarios.

A. Feasibility Checking for Two Clouds in a Single Slot

We first solve the TLM problem by considering a simplified scenario with two clouds in a single slot ($M = 2$, $K = 1$). Since there is only a single slot, any feasible solution needs to satisfy all users' requests in this slot. Therefore our goal in this scenario is to find a feasible solution that has the minimum cost. Our basic idea to find a feasible solution is to greedily let users download data from a cheaper cloud until the cheaper cloud is fully loaded. Here we assume that for any user $U_j \in U$ the needed data size W_j is lower than the total

Algorithm 1 Greedy Algorithm

Inputs: Cloud sites C, Users U, Cloud capacity u_i, End-to-end capacity d_{ij}, Cost per unit G_i

Outputs: x_{ij}

```
1: //Without loss of generality we assume that $G_1 \leq G_2$
2: for all $j \in U$ do
3:     $x_{1j} \leftarrow \frac{min\{W_j, d_{1j}, b_j\}}{\omega}$, $x_{2j} \leftarrow \frac{W_j - x_{1j} \cdot w}{\omega}$
4: end for
5: if $\sum_{j=1}^{N} x_{2j} > u_2$ or $x_{1j} + x_{2j} > b_j$ then
6:     //Cannot find a feasible solution.
7:     No feasible solution, return.
8: else if $\sum_{j=1}^{N} x_{1j} < u_1$ and $\sum_{j=1}^{N} x_{2j} < u_2$ then
9:     //A feasible solution with minimal cost is found.
10:    return all $x_{ij}$.
11: else
12:    //Move $\delta$ workload from $C_1$ to $C_2$
13:    $l \leftarrow \sum_{j=1}^{N} x_{1j}$
14:    for $j = 1 : N$ do
15:        $\delta \leftarrow min\{x_{1j}, (min\{d_{2j}, b_j\} - x_{2j})\}$
16:        $x_{1j} \leftarrow (x_{1j} - \delta)$, $x_{2j} \leftarrow (x_{2j} + \delta)$, $l \leftarrow (l - \delta)$
17:        return all $x_{ij}$ if $l \leq u_1$.
18:    end for
19:    No feasible solution, return.
20: end if
```

available bandwidth of U_j, i.e. $W_j \leq \mathbf{min}\{(d_{1j} + d_{2j}), b_j\}$. Otherwise there is no feasible solution.

Algorithm 1 shows the detail of our greedy algorithm. Without loss of generality we assume that the cost of each cloud satisfies $G_1 \leq G_2$. We follow the next two steps to find a feasible solution. First, we preferentially assign the bandwidth of the cheaper cloud C_1 to all users as much as possible, as shown in line 2-4. If C_2 is overloaded after the assignment, or data chunks requested by user exceed the user bandwidth constraint b_j (line 5-7), there is no feasible solution. And if neither cloud nor user bandwidth is overloaded, a feasible solution can be found. (line 8-10).

Second, we consider the case where C_1 is overloaded but C_2 still has available capacity. We adjust the bandwidth assignment by gradually moving a portion of workloads from C_1 to C_2 until we find a feasible solution where both clouds have acceptable workloads (line 12-20). Algorithm 1 outputs the optimal solution in $O(N)$ in the two-cloud scenario, where N is the number of users.

B. Maximum Tail Minimization Algorithm

We further study a more complex scenario with multiple clouds in multiple slots, and design a polynomial time optimal algorithm, named *Maximum Tail Minimization Algorithm (MTMA)* to solve the TLM problem. The key idea of MTMA is to search the optimal solution by performing the following two steps: (1) transforming the TLM problem to the *Minimum Cost Maximum Flow (MCMF)* problem, and (2) leveraging the binary search to explore the minimum latency under the given cost constraint.

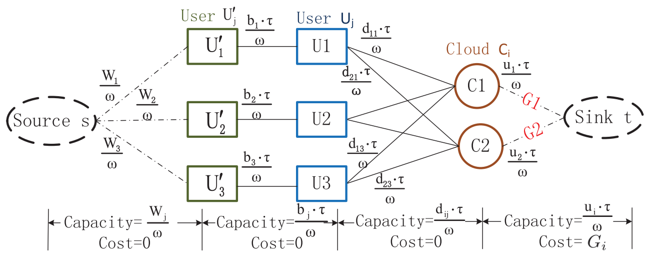

User U'_j
User U_j
Cloud C_i
$\frac{b_1 \cdot \tau}{\omega}$
$\frac{b_2 \cdot \tau}{\omega}$
$\frac{b_3 \cdot \tau}{\omega}$
$\frac{W_1}{\omega}$
$\frac{W_2}{\omega}$
$\frac{W_3}{\omega}$
$\frac{d_{11} \cdot \tau}{\omega}$
$\frac{d_{21} \cdot \tau}{\omega}$
$\frac{d_{13} \cdot \tau}{\omega}$
$\frac{d_{23} \cdot \tau}{\omega}$
$\frac{u_1 \cdot \tau}{\omega}$
$\frac{u_2 \cdot \tau}{\omega}$
Source s
Sink t
U'_1
U'_2
U'_3
U1
U2
U3
C1
C2
G1
G2
Capacity= $\frac{W_j}{\omega}$ Cost=0
Capacity= $\frac{b_j \cdot \tau}{\omega}$ Cost=0
Capacity= $\frac{d_{ij} \cdot \tau}{\omega}$ Cost=0
Capacity= $\frac{u_i \cdot \tau}{\omega}$ Cost= G_i

Fig. 8. Problem transformation: single slot.

Transforming TLM to MCMF: We first transform the original TLM problem to the MCMF problem by constructing a flow graph $G(V, E)$. We start the transformation from a simple case with only one single slot as shown in Figure 8. Let the vertex set V include all elements in the cloud set C, the user set U, and a copy of U, denoted as U'. In addition, we add a source vertex s and a sink vertex t to V. We build four groups of directed edge: (1) from s to each user in U'; (2) from each user in U' to its corresponding mirror in U; (3) from each user in U to all clouds in C; (4) from each cloud in C to t.

Each edge in E has two properties: (1) the link capacity, and (2) the cost of each data chunk delivered by the link. We then assign the link capacity and cost for each edge in the flow graph. First, for edges bridging the source s and U'_j, we assign the link capacity $\frac{W_j}{\omega}$ and the cost is set to zero. Second, for edges linking U' and U, we set the link capacity to $(b_j \cdot \tau)/\omega$ and set the cost to zero. For edges between U_j and C_i, we set the link capacity to the end-to-end bandwidth constraint d_{ij}, while setting the cost to zero. In addition, we set the capacity and cost of the edge between C_i and the sink t to u_i and G_i respectively.

Hence we obtain the flow graph as shown in Figure 8 and the original TLM problem is then transformed to the MCMF problem: given a flow graph $G(V, E)$ where each edge has a capacity and a cost for delivering data chunks, how to assign the number of data chunks in every edge to deliver all data chunks from the source s to the sink t, while minimizing the total cost. If the minimum cost of the MCMF problem is lower than the cost constraint, then the feasible assignment for the MCMF problem is the optimal solution for the original TLM problem that minimizes the latency under the cost constraint.

Next we extend the single-slot flow graph in Figure 8 to work in the multi-slot scenario. We extend the vertex set V by making C_i and U_j in slot k as new vertexes denoted as C_{ik} and U_{jk}, as shown in Figure 9. Let D denote the maximum fair user-perceived latency of all users, i.e. $D_j \leq D$. Thus the request of U_j completes after A_j and before $A_j + D \cdot (\frac{W_j}{\omega})^{\lambda}$. We construct the flow graph in multi-slot scenario following the methodology introduced above, but we only build an edge between C_{ik} and user U_{jk} in slots between A_j and $A_j + D \cdot (\frac{W_j}{\omega})^{\lambda}$. We set the link capacity of edges from s to U'_j to $\frac{W_j}{\omega}$, while setting the cost to zero. For edges from U'_j to U_{jk} we set the capacity to $\frac{b_j \cdot \tau}{\omega}$ and set the cost to zero. Similarly, for edges from U_{jk} to C_{ik} we set the link capacity to $\frac{d_{ijk} \cdot \tau}{\omega}$ and

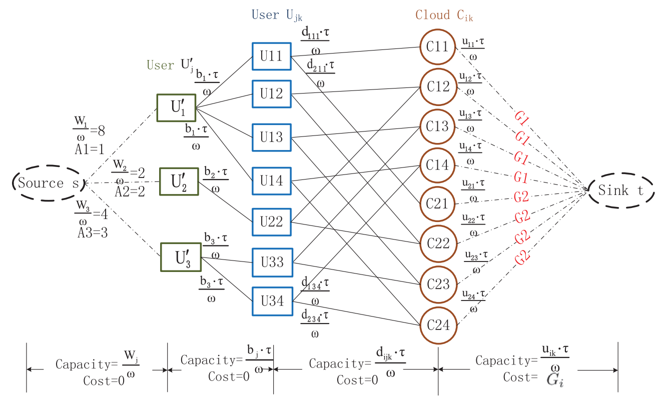


Fig. 9. Problem transformation: multiple slots.

set the cost to zero. Finally for edges from C_{ik} to t, the link capacity is set to $\frac{u_{ik}\cdot\tau}{\omega}$ and the cost is set to G_i. Therefore, as shown in Figure 9, let $G_D(V,E)$ denote the constructed flow graph, our TLM problem in the multiple-slot scenario is then transformed to the problem of finding the minimum D guaranteeing that $G_D(V,E)$ has a feasible MCMF solution.

Searching the Optimal Solution: We then perform binary search on $G_D(V,E)$ to find the minimum D and the corresponding data chunk assignment (i.e. x_{ijk}) on each edge. Initially we set the minimum $D_{left} = 0$ and set D_{right} to be a large enough delay D_{MAX}, e.g. the largest fair user-perceived latency observed in history. We search the minimum D between D_{left} and D_{right} that generates a feasible solution for the corresponding MCMF problem. Algorithm 2 shows the design details of MTMA. After initialization (line 1-2), in each iteration of the binary search we first construct a flow graph $G_D(V,E)$ and solve the corresponding MCMF problem (line 4-5). The optimal D is found if the flow graph has a feasible solution, while the total cost is less than the constraint f under D but exceeds f under $D-1$ (line 6-17). Our MTMA can optimally solve the problem in polynomial time. The complexity of MTMA is $O(VE\cdot log(V)\cdot log(VS)\cdot log(D_{MAX}))$, where $O(log(D_{MAX}))$ is the complexity of the binary search. $O(VE\cdot log(V)\cdot log(VS))$ is the complexity of solving the MCMF problem, where S is the maximum cost of edges. The runtime of MTMA is $O(K^2MN(M+N)log(K(M+N))log(KG(M+N))logD)$ where G is the maximum cost of all edges.

C. RHC-Based MTMA

Ideally, the above MTMA can optimally solve the TLM problem in polynomial time. However the MTMA suffers from a limitation that all related knowledge (e.g. cloud bandwidth u_i and request arrive time A_j) in a scheduling period (e.g. $[k, k+K-1]$) should be known in advance when we run MTMA for scheduling. In practice, such perfect knowledge is not available, making it difficult to find such optimal solutions using MTMA.

To overcome the above limitation, we develop the *Receding Horizon Control based (RHC-based) MTMA* by extending the MTMA to work at online mode. The key idea of the RHC-based MTMA is: while the perfect knowledge in an entire future schedule period may not be available, it is possible that reasonably accurate throughput prediction can be instead obtained for a short horizon to the future, e.g. in $[k, k+H-1]$, $H < K$. The intuition here is that the network conditions are reasonably stable on short timescales and usually do not drastically change during a short horizon [28]. Based on this insight, we can run MTMA using the prediction in the horizon to obtain all assignments in $[k, k+H-1]$, and then move the horizon forward to $[k+1, k+H]$. This scheme is known as receding horizon control (RHC) [29], and is widely used in different domains, ranging from industrial control to navigation. The general benefits of RHC are that RHC can utilize predictions to optimize a complex control objective online in a dynamical system under constraints.

Algorithm 2 Maximum Tail Minimization Algorithm

Inputs: Cloud sites C, Users U, Cloud capacity u_{ik}, End-to-end capacity d_{ijk}, User capacity b_j, Request data amount W_j, Cost per chunk G_i, Cost constraint f

Outputs: D, x_{ijk}

```
1: $D_{left} \leftarrow 0, D_{right} \leftarrow D_{MAX}$
2: $D \leftarrow \frac{D_{left}+D_{right}}{2}$
3: while TRUE do
4:   Construct the graph $G_D(V,E)$ according to $D$, $C$, $U$, $u_{ik}$, $d_{ijk}$, $b_j$, $W_j$, $D_j$, $G_i$, $f$.
5:   Solve $G_D(V,E)$ by MCMF to get $\{x_{ijk}\}$.
6:   if $\{x_{ijk}\}$ does not exist then
7:     $D = \frac{D+D_{right}}{2}$
8:   else
9:     $cost(D) \leftarrow \sum_{i=1}^{M}\sum_{j=1}^{N}\sum_{k=A_j}^{K} x_{ijk}\cdot G_i$
10:    if $cost(D) \le f$ then
11:      if $cost(D-1) > f$ then
12:        break, return $\{x_{ijk}\}$, $D$.
13:      else
14:        $D = \frac{D+D_{left}}{2}$
15:      end if
16:    else
17:      $D = \frac{D+D_{right}}{2}$
18:    end if
19:  end if
20: end while
```

Algorithm 3 illustrates the details of the RHC-based MTMA. Specifically, given that a receding horizon covers H sequential time slots, the RHC-based MTMA iteratively executes four key steps in each slot k:

- *Predicting (lines 3-5):* Predict the bandwidth in the current horizon $[k, k+H-1]$. Note that our goal in this paper is not to design a prediction mechanism and hence we rely on existing approaches. Naturally, improving the accuracy of the prediction will improve the gains achieved via RHC-based MTMA;
- *Checking (lines 6-9):* To accomplish robustness and efficiency, in each iteration we check the accuracy of prediction in the last slot $k-1$. If the last prediction in slot $k-1$ matches the observed real bandwidth in slot

Algorithm 3 Adaptative RHC-Based MTMA

```
Inputs: Cloud sites $C$, Users $U$, Current Cloud capacity $u_{i[1,k]}$, Current End-to-end capacity $d_{ij[1,k]}$, User capacity $b_j$, Request data amount $W_j$, Cost per chunk $G_i$, Cost constraint $f$
Outputs: $x_{ijk}$
1: Initialize
2: for $k = 1$ to $K$ do
3:     //Step 1: Predicting
4:     $\hat{d}_{ij[k,k+H-1]} = BandwidthDPred(d_{ij[1,k]})$
5:     $\hat{u}_{i[k,k+H-1]} = BandwidthUPred(u_{i[1,k]})$
6:     //Step 2: Checking
7:     if $Prediction_{k-1}$ matches the observed value in slot $k-1$, and there are no unscheduled requests then
8:         Go to step 4.
9:     end if
10:    //Step 3: Scheduling
11:    $R_{j[k,k+H-1]} = R_{j[k-1,k+H-2]} - R_{j(k-1)} + R_{jk}$
12:    Call MTMA with $R_{j[k,k+H+1]}, \hat{d}_{ij[k,k+H-1]}, \hat{u}_{i[k,k+H-1]}$ and $f_{avg} = f \cdot \frac{H}{K}$ to calculate $x_{ij[k,k+H-1]}$
13:    //Step 4: Applying
14:    Follow $x_{ijk}$ to download data for each user.
15: end for
```

$k-1$, and there are no unscheduled requests (e.g. new or remaining requests), we skip the scheduling step and go to the applying step directly. Note that there is a case where some requests for large files may not finish in one horizon, and these remaining requests will be scheduled next time when executing the scheduling step;

- *Scheduling (lines 10-12):* We first calculate the total number of requests from each user U_j in horizon $[k, k+H-1]$, denoted as $R_{j[k,k+H-1]}$ (line 11). $R_{j[k,k+H-1]}$ equals to the sum of unfinished requests $R_{j[k-1,k+H-2]} - R_{j(k-1)}$ which have not been downloaded yet after slot $k-1$, and new requests R_{jk} arriving in slot k. Secondly, given the predicted bandwidth in horizon $[k, k+H-1]$, we call MTMA to schedule all requests and obtain assignments x in $[k, k+H-1]$ (line 12). Note that we calculate the average cost constraint for each horizon ($f_{avg} = f \cdot \frac{H}{K}$) and use it as the input for MTMA. Since the scheduling step outputs the assignment for $[k, k+H-1]$, in next slots the scheduling step can be skipped if the bandwidth does not drastically change and all requests have already been scheduled;
- *Applying (lines 13-14):* Finally we apply the assignment in current slot k, i.e. for U_j, and start to download x_{ijk} data chunks from C_i.

Practical Bandwidth Predictor: Developing good predictors for different scenarios is beyond the scope of the paper. Building on insights from prior work, we use the harmonic mean of the observed throughput of the last 5 slots to estimate the bandwidth in current horizon, because this method is robust to outliers in bandwidth estimates [30]. Besides, for requests from new users without prior bandwidth knowledge, we estimate the bandwidth by calculating the harmonic mean of the observed throughput of other users from the same IP prefix, since Internet latencies to any particular data center are similar from all end-hosts in the same prefix [23].

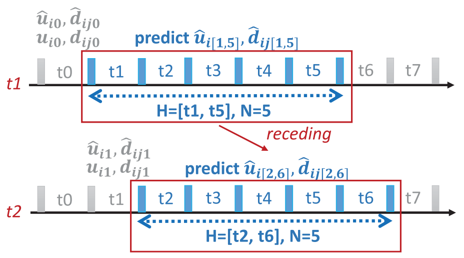

$\hat{u}_{i0}, \hat{d}_{ij0}$
u_{i0}, d_{ij0}
predict $\hat{u}_{i[1,5]}, \hat{d}_{ij[1,5]}$
t1
t0 t1 t2 t3 t4 t5 t6 t7
H=[t1, t5], N=5
receding
$\hat{u}_{i1}, \hat{d}_{ij1}$
u_{i1}, d_{ij1}
predict $\hat{u}_{i[2,6]}, \hat{d}_{ij[2,6]}$
t2
t0 t1 t2 t3 t4 t5 t6 t7
H=[t2, t6], N=5

Fig. 10. An example of how RHC-based MTMA moves horizon over time.

Workflow Inside TAILCUTTER*:* Next we explain the workflow of the RHC-based MTMA in the architecture of TAILCUTTER. In every slot the available bandwidth of each cloud and the end-to-end bandwidth are measured by the Measurement Agent. The Request Scheduler then collects the measurement results and predicts the bandwidth in the current horizon. After that the Request Scheduler checks the correctness of the last prediction and whether there remain unscheduled requests. If a drastic bandwidth variance is observed or there are unscheduled requests, the MTMA is invoked to compute new assignments which are later applied to each user.

Figure 10 plots an example to illustrate how RHC-based MTMA works over time. Assume the horizon covers five time slots, at slot t_0 the request scheduler predicts the available bandwidth $\hat{d}_{ij[1,5]}$ and $\hat{u}_{i[1,5]}$. If the prediction $\hat{d}_{ij0}$ and $\hat{u}_{i0}$ in last slot t_0 do not approximately match the observed bandwidth d_{ij0} and u_{i0} or there are unscheduled requests, then the Request Scheduler calculates the total number of unfinished and newly arrived requests of each user, and runs MTMA to obtain an assignment in $[t_1, t_5]$. Otherwise all users just follow the assignment made in the last schedule to download data. After that the horizon moves forward to $[t_2, t_6]$ in the next iteration.

This workflow has several qualitative advantages. First, the RHC-based MTMA leverages both bandwidth prediction and the cost-effectiveness of using multiple clouds in a principled way. Second, compared to the previous optimization in cloud CDNs [2], [3], our solution smooths our prediction error at each slot and is more efficient and robust to prediction errors. Specifically, by checking the correctness of the prediction over a moving horizon, large prediction errors for one particular request will have a lower impact on the performance.

V. PERFORMANCE EVALUATION

To evaluate the effectiveness of TAILCUTTER, we build the TAILCUTTER prototype on commercial cloud storage services and conduct a trace-driven evaluation using real-world traffic collected from a major ISP. In addition, we also conduct an extensive simulation to evaluate the performance of TAILCUTTER at scale.

A. TAILCUTTER *Implementation*

We implement the TAILCUTTER server over five data centers across Amazon AWS and Microsoft Azure ($M = 5$),

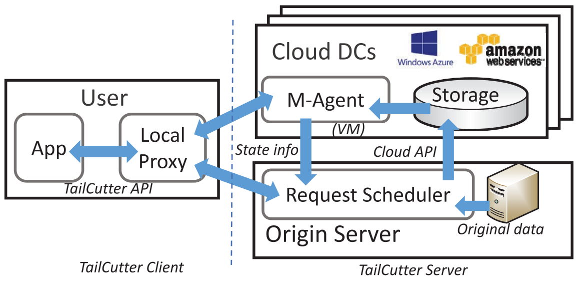


Fig. 11. TAILCUTTER's implementation on multiple cloud providers.

and we deploy the client counterpart on 12 geo-distributed VMs, as shown in Figure 11. We discuss the details of each key component in turns next.

Request Scheduler: We deploy the Request Scheduler in a Linux server with a 3.3 GHz Intel Octal Core CPU and 16 GB memory. We run our scheduling algorithm to manage bandwidth and make assignments for user requests based on the information collected from each Measurement Agent in every slot. The Request Scheduler stores access tokens of every cloud in a SQLite database and before the first scheduling period replicas are delivered to each cloud data center via the specific cloud API given by cloud providers.

Measurement Agent: We implement the Measurement Agent in an Amazon EC2 instance or a Windows Azure VM instance inside each cloud data center. Since the data in the storage service are delivered to the user relying on the VM instance, our Measurement Agent invokes cloud-specific APIs to fetch data from the storage service and uses `tcpdump` to collect network traces and calculate the bandwidth information, which is periodically sent to the Request Scheduler over HTTP.

Local Proxy: We implement the Local Proxy as a library in the end-host that exposes TAILCUTTER API to applications. To download a file, the Local Proxy issues a set of HTTP-GET requests to multiple cloud data centers following the assignment made by the Request Scheduler.

B. Experimentation Setup

Trace Collection: We collect a real-world data trace from QQ Xuanfeng on February 24, 2015 to evaluate the sensitivity of TAILCUTTER with different fairness factors. This trace contains 552 thousand flow records. Then we use a large scale real-world data trace to evaluate TAILCUTTER. The trace is generated from 99 collection points by a local major ISP on January 10, 2013. The trace data capture about 821 million flow records (about 2 Terabytes). Each record corresponds to the information of one flow which contains the user IP, server IP, flow time stamps, downloaded data size but without any personal data. We randomly pick workloads from 12 IPv4 subnetworks (/24) from our data trace as the workload of user requests and replay them on each VM in our experiment.

Methodology: We first use the real-world data trace from QQ Xuanfeng to evaluate the sensitivity of TAILCUTTER with different fairness factors. We further use the real-world data trace to evaluate our TAILCUTTER prototype from four aspects: (1) the ability in cutting tail latency; (2) the cost overhead; (3) the efficiency of executing the scheduling algorithm;

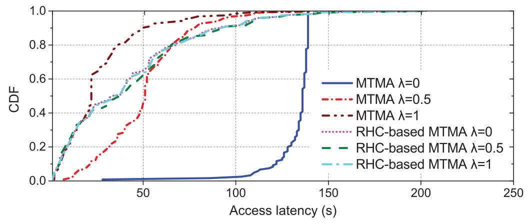


Fig. 12. User-perceived latency of TAILCUTTER with various fairness factor λ.

and (4) the cost-effectiveness in optimizing user-perceived tail latency. Then we conduct a large scale simulation to demonstrate the effectiveness of TAILCUTTER at scale. In our experiment, the scheduling period T is set to 1 hour and each slot lasts for 30 seconds (i.e. $K = 120$, and $\tau = 30s$). In our current implementation of the RHC-based MTMA we set the horizon length $H = 5$, and we set the cost overhead of each cloud according to the corresponding pricing policy of Amazon AWS and Microsoft Azure [31], [32].

Comparison: In our experiment we compare the performance of TAILCUTTER with other four solutions focusing on request scheduling in cloud CDNs: (1) *GRP* [2] which sends a single request to the nearest cloud to download data under the cost constraint. GRP can work as an offline or online scheduler, depending on whether the bandwidth and request knowledge of the entire period is known at the scheduling time; (2) *CosTLO* [3] that issues redundant requests to nearby clouds to reduce the tail latency and total cost overhead; (3) *Parallelism2*, which leverages the basic idea we explored in Section II-C to send two parallel requests to nearby clouds and download different parts of the replica to save the cost overhead; (4) *Parallelism4*, which leverages the basic idea of TAILCUTTER to simply send four parallel requests to nearby clouds but not in a cost-efficient manner, and download different parts of the replica to complete the downloading task. Note that Parallelism4 is a simplified version of TAILCUTTER which does not take the cost constraint into consideration.

C. Sensitivity of TAILCUTTER Fairness Factor λ

Figure 12 plots user-perceived latency of TAILCUTTER when we adopt different fairness factors λ. User-perceived latency of MTMA is much different from each other with different λ. However, user-perceived latency of RHC-based MTMA is almost similar to each other with different λ, because RHC-based MTMA divides the whole schedule to multiple small horizons and schedules the assignment in these small horizons dynamically.

When $\lambda = 0$, the objective of the TLM problem is to minimize the maximum user-perceived latency while satisfying total cost constraint. 100th percentile user-perceived latency when $\lambda = 0$ reduces by 23% compared to that when $\lambda = 1$. However 0-99th percentile user-perceived latency when $\lambda = 0$ is much higher than others obviously. This is because in this setting MTMA treats all users who request different sizes of replica equally, and it spends more bandwidth resources on the maximum of user-perceived latency, which may hurt the performance of most users. Therefore, adopting $\lambda = 0$ is more

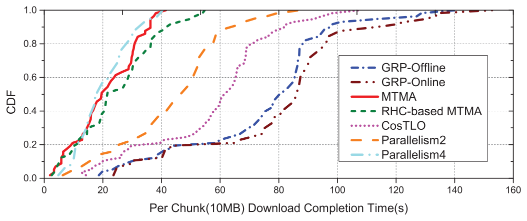


Fig. 13. Verification of TailCutter's ability to reduce fair user-perceived latency variance of each user.

suitable in specific traces where the size of different users is similar.

When $\lambda = 1$, the objective of the TLM problem is to minimize the maximum of per-chunk latency. The user-perceived latency of most users when $\lambda = 1$ outperforms others in this real-world data trace in view of different sizes of replica for different users. As a result, adopting $\lambda = 1$ is more suitable when there is a big difference of the size of replica among different users, which is common in most real-world data traces (e.g. Traces of TailCutter prototype and a large-scale simulation). Therefore we adopt $\lambda = 1$ in the following evaluation.

D. Ability to Reduce Access Latency

TailCutter is able to satisfy the cost overhead constraint and optimize fair user-perceived latency by properly scheduling user requests to different cloud data centers. We first evaluate TailCutter's ability to reduce tail latency and Figure 13 plots the user-perceived access latency under different scheduling mechanisms using the same data trace. For TailCutter we set the cost constraint to \$1700. We observe that TailCutter can effectively decrease fair user-perceived latency and also reduce the latency variance as compared to other alternatives. More specifically we made the following observations: (1) GRP gets the worst latency performance, in both offline and online modes. This is because GRP does not leverage multiple clouds to schedule user requests and hence some users suffer from high latency due to the congestion, server outages, etc.; (2) TailCutter, CosTLO, Parallelism2 and Parallelism4 can significantly reduce the 100th percentile access latency compared to GRP, since they explore multiple clouds to boost the latency performance; (3) our RHC-based MTMA reduces the 100th percentile latency of CosTLO and Parallelism2 by 45% and 32% respectively, because RHC-based MTMA issues requests to download different parts of the replica, and has a central view to manage the bandwidth for all requests to avoid the impact of congestion; (4) Parallelism4 reduces the 100th percentile latency of MTMA by 2.7%, because Parallelism4 leverages one key idea of TailCutter which is using multiple clouds and downloading different parts of replica, and leaves cost budget of the cloud provider out of consideration; (5) the MTMA outperforms other schedulers except for Parallelism4 because it leverages the perfect bandwidth knowledge in a scheduling period to obtain the optimal solution.

We further examine the performance of each scheduler by evaluating the access latency under different workloads. We define *request frequency* as the number of requests arriving in one slot and then divide our original trace into two categories: (1) *light workload*, in which the request frequency is less than 100 on average; and (2) *heavy workload*, containing more than 100 requests per slot on average. We then replay both workloads under different schedulers and Figure 14 shows the results of tail latencies.

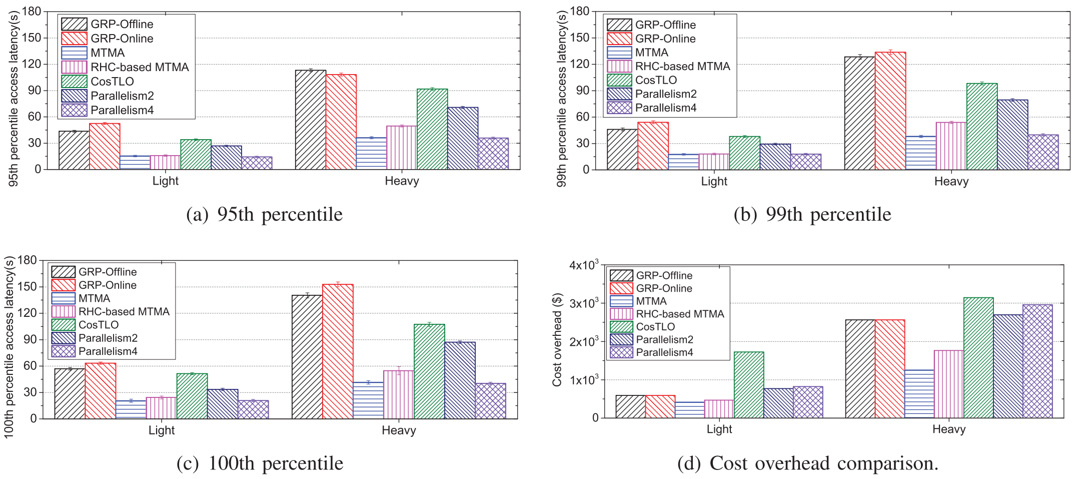
We observe that TailCutter's tail latency profile, even at the 100th percentile, degrades proportionally with the increase in system load. Specifically, for 95th, 99th and 100th percentile of latency in light workload, the MTMA in TailCutter outperforms other schedulers by up to 70.6%, while the RHC-based MTMA outperforms other alternatives by up to 69.8%. The latency reduction is mainly caused by leveraging multiple clouds and properly managing the bandwidth for each request. In addition, for 95th, 99th and 100th percentile of latency in the heavy workload, the MTMA in TailCutter outperforms other solutions by up to 73.3%, and the RHC-based MTMA in TailCutter outperforms others by up to 72.6%. Latency improvement of TailCutter in heavy workload is better than that in light workload. This is due to the fierce competition for bandwidth in the heavy workload, yet TailCutter stands in a central view to schedule requests and manage bandwidths to avoid congestions in the cloud.

Cost Comparison: Besides the latency performance which significantly affects user experience, the cost is also a very important metric for application providers. We compare the cost overhead of all schedulers in both light and heavy workloads. As shown in Figure 14(d), for the light workload, the MTMA in TailCutter reduces up to 75.9% cost overhead compared to other schedulers, and the RHC-based MTMA in TailCutter reduces up to 60.1% cost overhead. For the heavy workload, our MTMA and RHC-based MTMA can reduce up to 72.9% and 43.9% cost overhead respectively. This is because TailCutter is adaptive to the workloads and would properly choose the best clouds under the cost constraint. The cost overhead of CosTLO in both light and heavy workloads are much higher than others since it issues redundant requests to download data and inevitably involves additional cost due to the bandwidth waste.

E. Cost-Effectiveness

Since cloud prices vary considerably with bandwidth and region [33], TailCutter dynamically schedules requests and optimizes the fair user-perceived latency under the cost constraint. Intuitively, given a higher budget, TailCutter can achieve lower downloading latency but at a higher cost. To quantify how TailCutter reduces the tail latency under different budgets, we change the overall cost overhead constraint and evaluate the 100th percentile latencies.

Figure 15 depicts the 100th percentile latency as a function of the cost constraint f. At the higher end of the examined range of budget f, TailCutter significantly reduces the tail latency by making more users download data from a faster but more expensive cloud to speed up data transmission. As the budget f decreases, latency increases because TailCutter tries to find the cheaper clouds to serve users or select



(a) 95th percentile (b) 99th percentile (c) 100th percentile (d) Cost overhead comparison.

Fig. 14. Fair user-perceived tail latencies of different schedulers under light and heavy workloads.

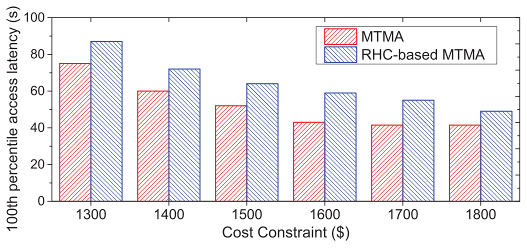


Fig. 15. Cost-efficiency evaluation in satisfying various cost constraints.

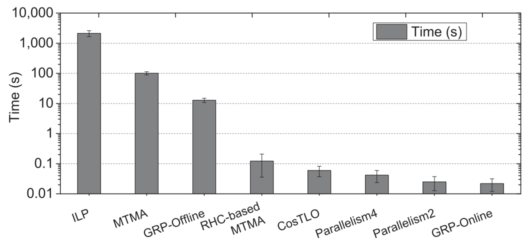


Fig. 16. Efficiency comparison of different scheduling algorithms.

fewer cloud data centers to process user requests. Because of the lower available bandwidth and resource competition, the latency increases. At the lower budget f, the latency of the RHC-based MTMA is slightly higher than the basic MTMA. This is because the RHC-based MTMA needs to schedule requests under available cost budget horizon by horizon, and it does not have the perfect knowledge of bandwidth and request distribution to schedule requests and manage bandwidth under the budget f.

F. Running Time in Practice

To evaluate the running time in practice, we run eight different schedulers: (1) straightforward scheduler by solving the ILP, (2) GRP offline, (3) MTMA, (4) RHC-based MTMA, (5) CosTLO, (6) Parallelism2, (7) Parallelism4 and (8) GRP online to make scheduling decisions for all the requests from 12 subnetworks in an hour. We repeat the scheduling under cost constraint ranging from \$1000 to \$4000, and calculate the average time of making a schedule. As shown in Figure 16, solving the original integer linear programming problem by CPLEX to schedule requests of a period consumes more than half an hour, which is not feasible in a real system. However, even though MTMA costs less computation time than ILP, consuming 101s is infeasible for a real-world workload monitor system. The RHC-based MTMA in TAILCUTTER only consumes 0.12s to obtain the scheduling results in a horizon, much more feasible for a real-world workload monitor system. In fact, many scheduling agents are needed in a large system, and Jiang et.al. propose solutions for agent selection in this situation [34].

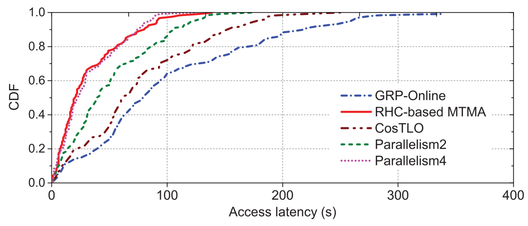
G. Simulation at Scale

In the above experiments we evaluate our TAILCUTTER prototype and demonstrate the effectiveness to reduce fair user-perceived tail latency while satisfying the cost constraint. We finally conduct a large scale simulation to evaluate TAILCUTTER at scale. We set 20 cloud data centers ($M = 20$) and select 4 workloads differing in the number of subnetworks from our trace data. These four workloads cover users from about 100, 200, 300 and 400 subnetworks respectively. The TAILCUTTER simulator is built using Matlab in around 1000 lines code. We set the bandwidth distribution and the pricing policy following a similar way in our prototype experimentation. Here we compare the results of five online alternatives in large-scale deployment scenarios.

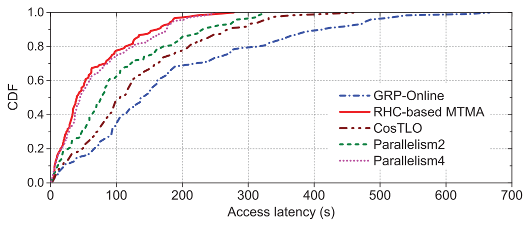
Figure 17 plots the CDF of fair user-perceived latency of all requests in different workload scales. The fair user-perceived latency increases as the scale of workload increases because of more resource competitions. We find that RHC-based MTMA reduces up to 58.9% 100th tail latency as compared to other schedulers. This result indicates that TAILCUTTER can wisely schedule user requests to different cloud data centers and avoid high tail latency when network suffers from high latency variance.

H. Bandwidth Utilization

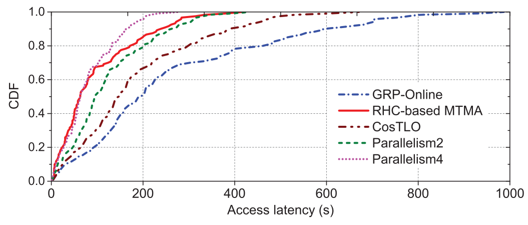
To further explore the advantage of TAILCUTTER, we plot the bandwidth utilization of all cloud servers in Figure 18. The bandwidth utilization of GRP keeps limited steadily, hence GRP has an obvious tail latency. CosTLO uses much



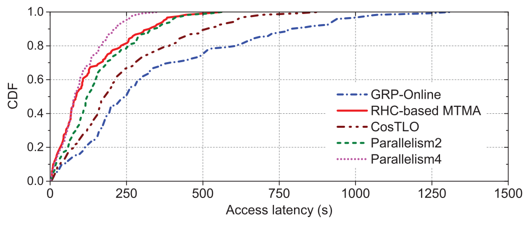
(a) # of Subnetwork = 100



(b) # of Subnetwork = 200



(c) # of Subnetwork = 300



(d) # of Subnetwork = 400

Fig. 17. CDF of fair user-perceived latency of replica with different system workloads.

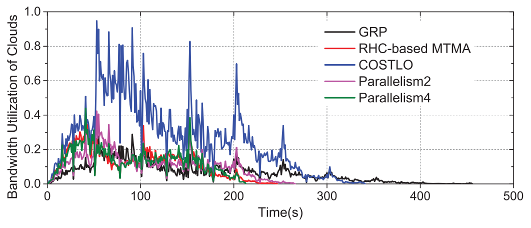


Fig. 18. Bandwidth utilization of different scheduling algorithms.

more bandwidth than all other solutions because it sends redundant requests to multiple clouds, as a result, the tail latency of CosTLO is reduced compared to GRP. Parallelism2, TAILCUTTER and Parallelism4 use more cloud bandwidth than GRP since they send parallel requests to multiple clouds, so they have better latency performance.

VI. DISCUSSION

Practicality of a Large-Scale Deployment: There are bandwidth measurements necessary for TAILCUTTER to schedule, so it is valuable to discuss whether TAILCUTTER can be used in large-scale deployment scenarios. Firstly, even though Request Scheduler seems to store a lot of states (bandwidth measurements of the user side b_j, from user to cloud d_{ijk}, and cloud side u_{ik}), actually it only stores limited states that are a horizon of latest bandwidth measurements. When new bandwidth measurements in the current slot come, Request Scheduler would delete old bandwidth measurements and add new results. Secondly, the basic idea of scheduling or controlling according to historical bandwidth measurements is widely used in many areas in industry, such as congestion control and video streaming. For example, BBR (Bottleneck Bandwidth and RTT) [35] which measures bottleneck bandwidth and round-trip propagation time to adjust packet send rate, is being deployed on Google and YouTube Web servers, substantially reducing latency; BBA (Buffer-Based Algorithms) [36] which chooses the video rate based on immediate past throughput achieves a significant performance improvement in Netflix. As a result, TAILCUTTER is possible to be used in large-scale deployment scenarios.

Fairness to Regular Users: Since TAILCUTTER involves a local proxy, it is necessary to discuss whether TAILCUTTER is fair to regular non-TAILCUTTER users. Here we will compare the cloud systems with TAILCUTTER and without TAILCUTTER. The influence of non-TAILCUTTER users in TAILCUTTER architecture depends on where the tail happens, and TAILCUTTER does not have a negative effect on most scenarios. When the bottleneck is in the user bandwidth of non-TAILCUTTER users or bandwidth from non-TAILCUTTER users to cloud, there is no difference whether TAILCUTTER is used. When the cloud which non-TAILCUTTER users request is not overloaded, there is no impact on these non-TAILCUTTER users. When the cloud which non-TAILCUTTER users request is overloaded, TAILCUTTER users would send parallel requests to balance the load of clouds, which even has a positive impact on these non-TAILCUTTER users.

VII. RELATED WORK

A considerable amount of research has been done on optimizing the performance of cloud storage services.

Optimizing QoS in Cloud CDNs: A lot of research has been done for optimizing the QoS in cloud CDNs [2]–[9], [37]–[40]. Authors of [10] and [11] proposed algorithms to optimize total storage and update cost. They used the assumption that requests can be issued from any node, but ignored retrieval cost. All solutions described above are static in nature in that they simply assume that the link quality between the client and the cloud is constant. They ignore the latency variance within cloud data centers or over the Internet.

Improving Performance Inside Cloud Data Centers: There are many recent studied working on redesigning storage systems and data centers to offer bandwidth guarantees to tenants [41], ensuring predictable completion times for TCP flows [14], [42], [43], or meeting tail latency Service Level Objectives (SLOs) inside data centers [44]–[49]. IOFlow [44] introduces a QoS architecture for controlling congestion via rate limiting and prioritization of storage and network I/O. PriorityMeister [48] addresses how to automatically configure priorities and rate limits to meet tail latency SLOs using a Deterministic Network Calculus (DNC) analysis. SNC-Meister [45] shows significant improvements in admission control when using the probabilistic analysis. These studies focus on optimizing the tail latency for small requests within data centers, and they complement our study since TAILCUTTER instead satisfies cost constraint for applications deployed on the cloud and optimizes user-perceived latency for larger requests. In addition, several recent works study similar scheduling problems but in the context of big data systems. TetriSched [50] introduces a new cluster scheduler that optimizes when and where to run jobs so as to improve performance in heterogeneous clusters. Rayon [51] investigates the reservation-based scheduling to guarantee stringent SLAs of jobs. Our scenario is targeted at cloud CDNs, and therefore makes many domain-specific choices (e.g. leveraging multiple clouds and considering the price factor) to optimize user-perceived latency while meeting the given cost constraint.

Reducing Latency Variance of Cloud Services: The approach of issuing multiple requests to different clouds to reduce tail latency has been considered previously [7], [19], and Shah et al. [52] characterize the settings under which redundant requests help, and design scheduling policies that employ redundant-requesting to reduce latency. In [53], Joshi et al. analyze how redundancy affects the latency and propose a general redundancy strategy for an arbitrary service time distribution. But the focus has primarily been on understanding the implications of redundancy on system load. In contrast, our work formulates and addresses the problem of how to issue multiple requests to different cloud data centers to download different parts of replica in order to reduce fair use-perceived latency under the cost constraint. The idea of reducing latency using coded download via multiple parallel data transmissions has been explored in previous studies. Shah et al. [54] characterize the latency performance of MDS queues and provide upper and lower bounds for First-Come-First-Served (FCFS) scheduling policies. In [55], Chen et al. propose a queueing architecture leveraging the coding redundancy inherent in cloud storage systems and prove that a simple greedy policy is delay-optimal among all on-line scheduling schemes. Sun et al. [56] propose low-complexity thread scheduling policies for several important classes of data downloading time distribution and show significant improvements in delay performance compared to FCFS. These studies focus on coded download which downloads the same size data chunks from different storage nodes. Our scenario allows users to download any number of file chunks from different clouds under the cost constraint in cloud CDNs. Some researchers have started to study delay-optimal load balancing scheme [57], optimizing data freshness and throughput [58]–[60], as well as network control without Channel State Information (CSI) [61], [62]. Some application providers such as Facebook use in-memory caching of data to reduce tail latencies [22]. However, caching at a single data center cannot tackle latency variance on Internet paths, and not all application providers will be able to afford caches at multiple data centers that can accommodate enough data to reduce 100th-percentile GET latencies.

Cloud Measurement Studies: Previous studies have compared the performance offered by different cloud providers [63], [64] and studied application deployments on the cloud [65]. Moreover, Bodik et al. [66] focused on characterizing and modeling spikes in application workloads. All these previous work complement our work. Furthermore, our measurement in this paper demonstrates that the tail latency problem inside traditional cloud data centers also exists in cloud CDNs, and we further highlight that the high tail latency is caused by inevitable reasons within cloud data centers and over the Internet.

Task Scheduling in Cloud Computing: The performance of cloud services (e.g. cloud CDN) heavily depends upon the scheduling of tasks. Existing works have studied the problem of optimizing the execution time of all tasks (i.e. makespan) in cloud services [67], [68]. However, existing works focus on general scheduling methods on cloud computing environments, and do not take the cost constraint into consideration. Instead, TAILCUTTER focuses on the user-perceived latency of each user instead of the overall execution time of all users, and optimizes the tail latency under the cost constraint.

VIII. CONCLUSION

In this paper, we identify, formulate and address the high user-perceived latency problem in cloud CDNs. Specifically, we measure and analyze the latency performance of cloud storage services, finding that the high tail latency problem indeed exists in cloud CDNs. We then formulate the problem of how to minimize the fair tail latency in the network condition where high latency variance is inevitable. To address the high tail latency problem we propose and implement TAILCUTTER, a request scheduling mechanism that issues parellel requests to different cloud data centers to reduce tail latency. We implement TAILCUTTER in modern cloud data centers and extensive evaluations using real world data traces show that TAILCUTTER is able to cut up to 58.9% tail latency in cloud CDNs.

REFERENCES

[1] *Public/Private Cloud Storage Market*. Accessed: 2019. [Online]. Available: http://www.marketsandmarkets.com/PressReleases/cloud-storage.asp

[2] F. Chen, K. Guo, J. Lin, and T. L. Porta, "Intra-cloud lightning: Building CDNs in the cloud," in *Proc. INFOCOM*, Mar. 2012, pp. 433–441.

[3] Z. Wu, C. Yu, H. V. Madhyastha, and U. Riverside, "Costlo: Cost-effective redundancy for lower latency variance on cloud storage services," in *Proc. NSDI*. Berkeley, CA, USA: USENIX, 2015, pp. 1–16.

[4] G. Joshi, Y. Liu, and E. Soljanin, "On the delay-storage trade-off in content download from coded distributed storage systems," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 5, pp. 989–997, May 2014.

[5] Y. Wu *et al.*, "Scaling social media applications into geo-distributed clouds," in *Proc. INFOCOM*, Mar. 2012, pp. 684–692.

[6] D. Wang, G. Joshi, and G. Wornell, "Efficient task replication for fast response times in parallel computation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, pp. 599–600, 2014.

[7] A. Vulimiri *et al.*, "Low latency via redundancy," in *Proc. CoNEXT*, 2013, pp. 283–294.

[8] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker, "More is less: Reducing latency via redundancy," in *Proc. HotNets*, 2012, pp. 13–18.

[9] K. Gardner *et al.*, "Reducing latency via redundant requests: Exact analysis," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 1, pp. 347–360, 2015.

[10] X. Tang and J. Xu, "Qos-aware replica placement for content distribution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 10, pp. 921–932, Oct. 2005.

[11] H. Wang, P. Liu, and J.-J. Wu, "A QoS-aware heuristic algorithm for replica placement," in *Proc. 7th IEEE/ACM Int. Conf. Grid Comput.*, 2006, pp. 96–103.

[12] G. Rodolakis, S. Siachalou, and L. Georgiadis, "Replicated server placement with QoS constraints," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 10, pp. 1151–1162, Oct. 2006.

[13] J. Broberg, R. Buyya, and Z. Tari, "MetaCDN: Harnessing 'Storage Clouds' for high performance content delivery," *J. Netw. Comput. Appl.*, vol. 32, no. 5, pp. 1012–1022, 2009.

[14] D. Zats *et al.*, "DeTail: Reducing the flow completion time tail in datacenter networks," in *Proc. SIGCOMM*, 2012, pp. 139–150.

[15] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 63–74, 2011.

[16] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[17] *Xuanfeng Offline Downloading System*. Accessed: 2017. [Online]. Available: http://xf.qq.com

[18] Z. Li *et al.*, "Offline downloading in China: A comparative study," in *Proc. IMC*, 2015, pp. 473–486.

[19] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[20] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proc. NSDI*. Berkeley, CA, USA: USENIX, 2015, pp. 1–16.

[21] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *Proc. NSDI*, 2013, pp. 329–341.

[22] R. Nishtala *et al.*, "Scaling memcache at facebook," in *Proc. NSDI*. Berkeley, CA, USA: USENIX, 2013, pp. 385–398.

[23] H. V. Madhyastha *et al.*, "iPlane: An information plane for distributed services," in *Proc. USENIX OSDI*. Berkeley, CA, USA, 2006.

[24] R. Kapoor *et al.*, "Capprobe: A simple and accurate capacity estimation technique," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 67–78, 2004.

[25] *Alibaba Cloud Data Transfer Pricing*. Accessed: 2019. [Online]. Available: https://www.alibabacloud.com/product/cdn/pricing

[26] *AWS Data Transfer Pricing*. Accessed: 2019. [Online]. Available: https://aws.amazon.com/govcloud-us/pricing/data-transfer/

[27] *Cplex Optimizer*. Accessed: 2019. [Online]. Available: https://www-01.ibm.com/software/

[28] Y. Zhang and N. Duffield, "On the constancy of Internet path properties," in *Proc. 1st ACM SIGCOMM Workshop Internet Meas.*, 2001, pp. 197–211.

[29] E. F. Camacho and C. B. Alba, *Model Predictive Control*. London, U.K.: Springer, May 2007.

[30] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive," in *Proc. CONEXT*, 2012, pp. 326–340.

[31] *Amazon S3 Pricing*. Accessed: 2019. [Online]. Available: https://aws.amazon.com/cn/s3/pricing/

[32] *Microsoft Azure Pricing*. Accessed: 2019. [Online]. Available: https://azure.microsoft.com/en-us/pricing/

[33] *Alibaba CDN Pricing*. Accessed: 2019. [Online]. Available: https://www.alibabacloud.com/product/cdn/pricing/

[34] J. Jiang *et al.*, "Via: Improving Internet telephony call quality using predictive relay selection," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 286–299.

[35] N. Cardwell *et al.*, "BBR: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 50-20–50-53, Oct. 2016.

[36] T.-Y. Huang *et al.* "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 187–198, 2015.

[37] Y. Huang, Z. Li, G. Liu, and Y. Dai, "Cloud download: Using cloud utilities to achieve high-quality content distribution for unpopular videos," in *Proc. MM*, 2011, pp. 213–222.

[38] Z. Li *et al.*, "Cloud transcoder: Bridging the format and resolution gap between Internet videos and mobile devices," in *Proc. NOSSDAV*, 2012, pp. 33–38.

[39] Z.-H. Li, G. Liu, Z.-Y. Ji, and R. Zimmermann, "Towards cost-effective cloud downloading with tencent big data," *J. Comput. Sci. Technol.*, vol. 30, no. 6, pp. 1163–1174, 2015.

[40] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "QuickSync: Improving synchronization efficiency for mobile cloud storage services," in *Proc. MobiCom*, 2015, pp. 592–603.

[41] H. Ballani *et al.*, "Chatty tenants and the cloud network sharing problem," in *Proc. NSDI*. Berkeley, CA, USA: USENIX, 2013, pp. 171–184.

[42] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 50–61, 2011.

[43] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proc. SIGCOMM*, 2012, pp. 127–138.

[44] E. Thereska *et al.*, "IOFlow: A software-defined storage architecture," in *Proc. SOSP*, 2013, pp. 182–196.

[45] T. Zhu, D. S. Berger, and M. Harchol-Balter, "SNC-Meister: Admitting more tenants with tail latency SLOs," in *Proc. SoCC*, 2016, pp. 374–387.

[46] K. Jang, J. Sherry, H. Ballani, and T. Moncaster, "Silo: Predictable message latency in the cloud," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 435–448, 2015.

[47] M. P. Grosvenor *et al.*, "Queues don't matter when you can jump them!" in *Proc. NSDI*, 2015, pp. 1–15.

[48] T. Zhu *et al.*, "Prioritymeister: Tail latency QoS for shared networked storage," in *Proc. SoCC*, 2014, pp. 1–14.

[49] S. Zhang, L. Huang, M. Chen, and X. Liu, "Proactive serving decreases user delay exponentially: The light-tailed service time case," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 708–723, Apr. 2017.

[50] A. Tumanov *et al.*, "Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proc. EuroSys*, 2016, p. 35.

[51] C. Curino *et al.*, "Reservation-based scheduling: If you're late don't blame us!" in *Proc. SoCC*, 2014, pp. 1–14.

[52] N. B. Shah, K. Lee, and K. Ramchandran, "When do redundant requests reduce latency?" *IEEE Trans. Commun.*, vol. 64, no. 2, pp. 715–722, Feb. 2016.

[53] G. Joshi, E. Soljanin, and G. Wornell, "Efficient redundancy techniques for latency reduction in cloud systems," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 2, p. 12, May 2017.

[54] N. B. Shah, K. Lee, and K. Ramchandran, "The mds queue: Analysing the latency performance of erasure codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun./Jul. 2014, pp. 861–865.

[55] S. Chen *et al.*, "When queueing meets coding: Optimal-latency data retrieving scheme in storage clouds," in *Proc. IEEE INFOCOM*, Apr./May 2014, pp. 1042–1050.

[56] Y. Sun *et al.*, "Provably delay efficient data retrieving in storage clouds," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr./May 2015, pp. 585–593.

[57] X. Zhou *et al.*, "Designing low-complexity heavy-traffic delay-optimal load balancing schemes: Theory to algorithms," Oct. 2017, *arXiv:1710.04357*. [Online]. Available: https://arxiv.org/abs/1710.04357

[58] Y. Sun *et al.*, "Update or wait: How to keep your data fresh," *IEEE Trans. Inf. Theory*, vol. 63, no. 11, pp. 7492–7508, Nov. 2017.

[59] A. M. Bedewy, Y. Sun, and N. B. Shroff, "Optimizing data freshness, throughput, and delay in multi-server information-update systems," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2016, pp. 2569–2573.

[60] A. M. Bedewy, Y. Sun, and N. B. Shroff, "Age-optimal information updates in multihop networks," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2017, pp. 576–580.

[61] Y. Sun, C. E. Koksal, S.-J. Lee, and N. B. Shroff, "Network control without CSI using rateless codes for downlink cellular systems," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1016–1024.

[62] Y. Sun, C. E. Koksal, K.-H. Kim, and N. B. Shroff, "Scheduling of multicast and unicast services under limited feedback by using rateless codes," in *Proc. IEEE INFOCOM*, Apr./May 2014, pp. 1671–1679.

[63] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: Comparing public cloud providers," in *Proc. IMC*, 2010, pp. 1–14.

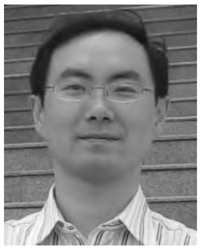
[64] X. Luo *et al.*, "Characterizing mobile*-box applications," *Comput. Netw.*, vol. 103, pp. 228–239, Jul. 2016.

[65] K. He *et al.*, "Next stop, the cloud: Understanding modern Web service deployment in EC2 and azure," in *Proc. IMC*, 2013, pp. 177–190.

[66] P. Bodik *et al.*, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proc. SoCC*, 2010, pp. 241–252.

[67] T. Mathew, K. C. Sekaran, and J. Jose, "Study and analysis of various task scheduling algorithms in the cloud computing environment," in *Proc. Int. Conf. Adv. Comput., Commun. Inform. (ICACCI)*, Sep. 2014, pp. 658–664.

[68] S. Nagadevi, K. Satyapriya, and D. Malathy, "A survey on economic cloud schedulers for optimized task scheduling," *Int. J. Adv. Eng. Technol.*, vol. 4, no. 1, pp. 58–62, 2013.

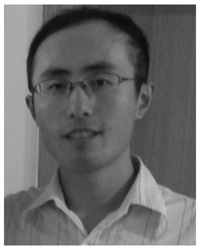


Yong Cui is currently a Ph.D. Professor with Tsinghua University. Having published more than 100 papers in refereed journals and conferences, he was a recipient of the Best Paper Award of ACM ICUIMC 2011 and WASA 2010. Holding more than 40 patents, he is one of the authors in RFC 5747 and RFC 5565 for his proposal on IPv6 transition technologies. He is a Council Member of the China Communication Standards Association and the Co-Chair of IETF IPv6 Transition WG Softwire.

Ningwei Dai received the B.Eng. degree in communication engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2015. She is currently pursuing the master's degree with the Department of Computer Science and Technology, Tsinghua University, Beijing. Her research interests include networking and cloud computing.

Zeqi Lai received the bachelor's degree in computer science from the University of Electronic Science and Technology of China in 2009. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Technology, Tsinghua University, China, under the supervision of Prof. Y. Cui. His research interests include networking and cloud computing.

Minming Li received the Ph.D. degree. He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong. His research interests include algorithms' design and analysis, combinatorial optimization, scheduling, key management, and algorithmic game theory. He received the City University of Hong Kong Teaching Excellence Award in 2011–2012.



Zhenhua Li (M'14) received the B.Sc. and M.Sc. degrees from Nanjing University in 2005 and 2008, respectively, and the Ph.D. degree from Peking University in 2013, all in computer science and technology. He is currently an Associate Professor with the School of Software, Tsinghua University. His research areas cover cloud computing/storage/download, big data analysis, content distribution, and mobile Internet.

Yuming Hu received the bachelor's degree in computer science from Tongji University, China, in 2017. He is currently pursuing the master's degree with the Department of Computer Science and Technology, Tsinghua University, China. His research interests include networking and cloud computing.

Kui Ren (F'16) is currently a Professor and the Director of the Institute of Cyberspace Research, Zhejiang University. His current research interests include cloud and outsourcing security, wireless and wearable system security, and human-centered computing. He currently serves as an Associate Editor for TMC, TIFS, IoT-J, TSG, *Pervasive and Mobile Computing*, and *The Computer Journal*. He is a member of the ACM and a Distinguished Lecturer of the IEEE Vehicular Technology Society.

Yuchi Chen received the bachelor's and master's degrees in computer science from the University of Electronic Science and Technology of China. He is currently pursuing the Ph.D. degree with the School of Computing Science, Simon Fraser University, BC, Canada, under the supervision of Prof. J. Liu. His research interests include networking and cloud computing.