When Queueing Meets Coding: Optimal-Latency Data Retrieving Scheme in Storage Clouds

Shengbo Chen
Department of ECE
Ohio State University
chens@ece.osu.edu

Yin Sun
Department of ECE
The Ohio State University
suny@ece.osu.edu

Ulaş C. Kozat
DOCOMO Innovations, Inc.
Palo Alto, CA, 94304
kozat@docomoinnovations.com

Longbo Huang
IIIS, Tsinghua University
longbohuang@tsinghua.edu.cn

Prasun Sinha
Department of CSE
Ohio State University
prasun@cse.ohio-state.edu

Guanfeng Liang
DOCOMO Innovations, Inc.
Palo Alto, CA, 94304
gliang@docomoinnovations.com

Xin Liu
Department of CS
UC Davis
liu@cs.ucdavis.edu

Ness B. Shroff
Department of ECE and CSE
The Ohio State University
shroff@ece.osu.edu

***Abstract*—Storage clouds, such as Amazon S3, are being widely used for web services and Internet applications. It has been observed that the delay for retrieving data from and placing data into the clouds is quite random, and exhibits weak correlations between different read/write requests. This inspires us to investigate a key problem: can we reduce the delay by transmitting data replications in parallel or using powerful erasure codes?**

In this paper, we study the problem of reducing the delay of downloading data from cloud storage systems by leveraging multiple parallel threads, assuming that the data has been encoded and stored in the clouds using fixed rate forward error correction (FEC) codes with parameters (n, k). That is., each file is divided into k equal-sized chunks, which are then expanded into n chunks such that any k chunks out of the n are sufficient to successfully restore the original file. The model can be depicted as a multiple-server queue with arrivals of data retrieving requests and a server corresponding to a thread. However, this is not a typical queueing model because a server can terminate its operation, depending on when other servers complete their service (due to the redundancy that is spread across the threads). Hence, to the best of our knowledge, the analysis of this queueing model remains quite uncharted.

Real traces from Amazon S3 show that the time to retrieve a fixed size chunk is random and can be accurately approximated as an *i.i.d.* exponentially distributed random variable. We show that any work-conserving scheme is delay-optimal when $k = 1$. When $k > 1$, we find that a simple greedy scheme, which allocates all available threads to the head of line request, is delay optimal, which appears surprising.

I. INTRODUCTION

Cloud storage, an essential element of cloud computing, has been rapidly expanding, backed by technology giants, such as Amazon, Google, Microsoft, IBM, and Apple, as well as now popular startups, such as Dropbox, Rack-space, and NexGen. The total cloud storage market is expected to grow from $5.6 billion in 2012 to $46.8 billion by 2018 with a compound annual growth rate of 40.2% [1]. Cloud storage systems provide users with easy access, low maintenance, flexibility, and scalability.

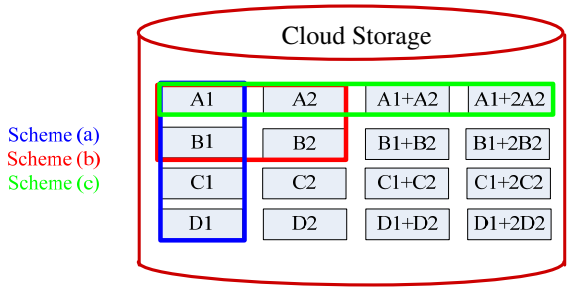

Fig. 1. An example of four files using FEC codes with parameters (4,2) in cloud storage and three thread allocation schemes

As in all storage systems, efficiency, reliability, and latency are critical requirements for cloud storage systems. Although simple duplications can be used for reliability, the most promising storage codes are forward-error-correction (FEC) codes, such as maximum-distance-separable (MDS) codes, which provide a better resiliency to erasures than duplication for a given amount of redundancy. In this paper, we consider FEC codes with fixed coding parameters (n, k), i.e., each file is divided into k equal-sized chunks, which are then expanded into n chunks such that any k out of n chunks are sufficient to successfully restore the k original chunks (hence, the file itself). Fig. 1 shows an example, where $n = 4$ and $k = 2$. In particular, four equal-length files A, B, C, and D are stored in the cloud using FEC codes with parameters (4,2). Each file $x \in \{A, B, C, D\}$ is partitioned into two equal-length chunks, $[x_1, x_2]$, and its four coded chunks, $[x_1, x_2, x_1 + x_2, x_1 + 2x_2]$, are stored in the cloud. A file can be retrieved from any two of its four chunks. In this manner, different levels of efficiency and reliability can be achieved by adjusting the parameters (n, k) of FEC, where we have $k > 1$ in general, while $k = 1$ simply means data duplication. Some online file storage companies have already

978-1-4799-3360-0/14/$31.00 2014 IEEE

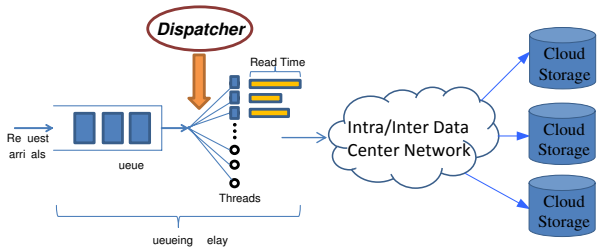

Fig. 2. The architecture for data retrieval through a multiple-thread dispatcher.

adopted such FEC codes, such as Wuala [2].

While reliability and efficiency have been carefully studied in cloud storage systems, its performance in terms of delay has received much less attention, even though delay is a critically important issue that significantly affects user experience and can play a major role in the widespread adoption of these systems. Measurement studies show that there exists a significant skew in network bound I/O performance [3]. Earlier evaluations of Amazon S3 indicate that the slowest 10-20% of read/write requests see more than 5× of the mean delays (e.g., see [4]). The experimental study of Amazon S3 in [3] shows a similar trend. In particular, the average delay of reading a 1MB file is 139*msec*, with 80% delay as 179*msec*, 95% as 303*msec*, 99.9% as 811*msec*, respectively. It has been shown that delay tail has a large impact on user experience and service provider revenue, e.g., every 500 ms extra delay in service will lead to a 1.2% user loss for Google and Amazon [5]. Thus, it is important to reduce the delay spread, i.e., cutting the long tail of the service time.

It has been revealed that the delay exhibits weak correlations between different read tasks [3], [4]. Thus, the delay can be reduced by simply transmitting data replications in parallel. This motivates us to investigate the critical issue of improving the system latency of data retrieval. In particular, a key question we ask is: *can we leverage the inherent redundancy provided by FEC for reliability to improve delay performance?* Assume that a user requests to download files, i.e., their encoded chunks, from the cloud storage systems with a dispatcher. The dispatcher schedules these downloading requests with a number of threads, where each thread can be used to retrieve one chunk each time. The number of threads that are assigned to the user is based on the function of the service that the user is willing to pay for — the higher the payment, the greater the number of threads allocated to the user. The model is depicted in Fig. 2. A typical application scenario is given as follows: an online game user needs to load and save (read and write) his data into the clouds using FEC codes. The user is allocated with a number of threads, where the number of threads is determined by how much the user is willing to spend.

In this paper, we focus on the delay of read requests (Note that our results can be also applied to write requests). We study the read delay induced by different thread allocation schemes, i.e., how the threads are assigned to retrieve distinct file chunks. To make the problem more concrete, consider the four encoded files A, B, C, and D, shown in Fig. 1. An user requesting these four files has four threads and can allocate these four threads using different allocation schemes, as shown in Fig. 1. In scheme (a), one thread is allocated to each file, i.e., four threads are allocated to download A1, B1, C1 and D1, depicted by the blue box. After downloading these chunks, the threads are then scheduled to download the second chunks of all four files. Scheme (b), depicted by the red box, provides parallelism where the two chunks of file A/B, i.e., A1, A2, B1 and B2, are retrieved in parallel. Scheme (c) further exploits the parallelism, and allocates all four threads to file A, each requesting a chunk, i.e., A1, A2, A1+A2 and A1+2A2, as depicted by the green box. As long as any two of the four chunks are retrieved, the dispatcher immediately *terminates* the other threads and allocates them to the next file. Clearly, the service delay of file A is minimized by scheme (c) since the file can be constructed from the first two successful chunks. On the other hand, scheme (a), compared to the previous two schemes, allows a greater number of parallel requests that can be served, i.e., four for scheme (a), two for scheme (b) and one for scheme (c). Therefore, there exists a tradeoff between the service time of a file and parallelism in data retrieval. And it is interesting to ask which scheme has the best delay performance.

In this paper, we resort to queueing theory and notice that the model can be depicted as a multiple-server queue with arrivals of data retrieving requests and a server corresponding to a thread. However, this is not a typical queueing model because a server can terminate its operation, depending on when other servers complete their service (due to the redundancy that is spread across the threads). According to our observations on real traces that are measured over Amazon S3 (see Section III), we approximate the downloading time for any individual thread as an *i.i.d.* exponentially distributed random variable. The randomness is because the communication within the cloud causes random delays.

In this paper, we make the following contributions:

- We propose a queueing architecture that leverages the coding redundancy inherent in cloud storage systems, to improve file retrieve latency performance. In particular, we present a new queueing model to study data retrieve latency, where each file is encoded and stored in the cloud using FEC codes.
- We rigorously analyze the delay performance under several schemes assuming that the service time to download a chunk is an exponentially distributed random variable. When $k = 1$, we show that any work-conserving scheme that fully utilizes all available threads is delay-optimal. For the case when $k > 1$, we prove that a simple greedy policy is delay-optimal. This is a somewhat surprising result, as delay optimality is rather strong in general, which also implies throughput optimality.
- We validate our theoretical results using both exponentially distributed service time distribution and real traces. We show that making use of inherent redundancy introduced by FEC can indeed reduce the delay and significantly

reduce the delay spread.

To the extent of our knowledge, we are the first to prove the delay-optimal scheme under the exponentially distributed downloading time in cloud storages system deploying FEC.

The organization of the paper is as follows. We discuss related work in Section II. In Section III, we present our measurement results over Amazon S3. Section IV describes the system model. In Sections V and VI, we present the results under the cases of $k = 1$ and $k > 1$, respectively. The simulation results are presented in Section VII. We then conclude our paper in Section VIII.

II. RELATED WORK

The use of coding to improve the efficiency of large-scale data storage systems has received significant attention [6]. For instance, the authors of [7] used interference alignment for reducing repair traffic for storage systems with erasure coding; in [8], the authors constructed explicit regenerating codes for achieving minimum bandwidth in distributed storage systems; in [9], the authors considered the problem of allocating capacity for optimal data storage. However, most of the existing works have focused mainly on utilizing coding to reduce repair bandwidth and storage capacity for storage systems (for more discussion, please see [10] and the references therein).

Recently, due to the increasing importance of service latency (i.e., delay) as a system performance metric, e.g., for Google and Amazon, every 500 ms extra delay in service will lead to a 1.2% user loss [5], researchers have started to study the effect of coding on content retrieval delay for data storage systems. The work [11] considered delivering a set of packets over a linear network with minimum delay. In [12], coding was used to reduce the blocking probability in storage networks. In [13], coding was used as a way to prevent service interruption. In [14] and [15], the authors investigated the performance of using multiple servers to download file replications without considering coding. The authors in [3] proposed a heuristic transmission control scheme by dynamically adjusting the coding parameters which demonstrates good delay performance in cloud storages.

The most related theoretical results that we know of are [16], [17], [18], which investigated how to assign the storage disks to serve the read requests for reducing delay. In [16], all the requests are put in a centralized queue and the authors showed that FEC codes can reduce the data retrieving delay compared to simple data replications. In [17], the authors proved that flooding requests to all storage disks, rather than a subset of disks, has a shorter data retrieving delay for the centralized queueing model. In [18], the requests are dispatched to multiple local queues at the storage disks and it was showed that coding reduces data retrieving time. Different from [16], [17], [18], we focus on the systems with limited downloading threads (bandwidth) and study how to allocate the downloading threads to exploit the storage redundancy and minimize the data retrieving delay. Since our system model is quite different from the existing studies, novel proof techniques are employed.

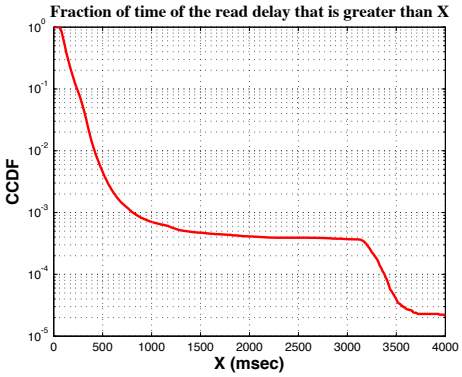

Fig. 3. CCDF of read delay for 1MB chunk.

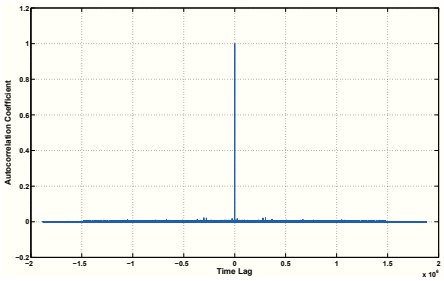

Fig. 4. Correlation between time ordered delay samples.

III. MEASUREMENT OVER AMAZON S3

In this section, we describe some measurement results of the read delay using Amazon S3 made by our coauthors from Docomo Innovations, Inc. [3]. Please refer to [3] for more details.

Fig. 3 plots the complementary cumulative distribution function (CCDF) of the delay for downloading a file of size 1MB. We can see that the downloading time indeed observes a wide spread, although the file size is the same across all experiments. Another observation is that despite the delay floors observed at very low percentiles (e.g., beyond the 99.9th percentile), up to 99th percentile, the CCDF is close to a linear term in delay and note that the y-axis is logarithmic. This indicates that an exponential distribution is a good approximation of the service time for most of the distribution, as proposed in the model.

Fig. 4 shows the autocorrelation coefficient between the service times of consecutive read tasks. Note that the mean delay is subtracted from the delay samples. We can see that there is negligible correlation between consecutive download delays. Based on these observations, we approximate the downloading times for any thread as *i.i.d.* exponentially distributed random variable.

IV. SYSTEM MODEL

As mentioned in the introduction, we consider a cloud storage system, where data is stored using FEC codes with coding parameters (n, k), i.e., each file has n chunks stored

in the clouds and any k out of the n chunks are sufficient to successfully restore the original file. We assume that the files in the cloud are homogeneous with same size and coding parameters, which also implies that all chunks are homogeneous with same size.

Data retrieval (read) is provided through a dispatcher with multiple threads, as shown in Fig. 2. When read requests arrive, they are first enqueued. The dispatcher then determines how to allocate threads for each request. For instance, in Fig. 2, there are three threads (rectangles) which are scheduled to download chunks from the cloud, while the other three threads (circles) are idle. Due to the service chosen by the user, the dispatcher can simultaneously keep at most L threads active in any time instant. We assume that the read requests arrive at the dispatcher with rate λ. Then, when a file is to be read from the cloud, the dispatcher schedules m ($k \leq m \leq n$) read tasks for distinct chunks of the file by activating a total of m threads (not necessarily simultaneously). Due to the use of FEC codes, the first k successful responses from the storage cloud will then be sufficient for completing the read operation. Hence, we assume that once k chunks are successfully downloaded, the other read tasks in progress can be terminated immediately.

For simplicity, we first assume $n \geq L + k - 1$. This assumption means that there are always enough distinct data chunks in the clouds, since it guarantees that there are still at least L distinct chunks in the cloud for the L threads to share given that $k-1$ chunks have been successfully downloaded. In this paper, we investigate the file retrieve latency under different thread allocation schemes, i.e., how should the dispatcher schedule the threads for serving the read requests.

A. Problem Formulation

We consider a time period $[0, T]$ and let $\mathcal{N} = \{1, 2, 3, \cdots, i, \cdots N_T\}$ denote the set of read request arrivals during this period, where i denotes the i-th arrival and N_T is the total number of request arrivals during the period. We denote n_i as the number of threads that the dispatcher allocates for the request i. Clearly, $k \leq n_i \leq n$. We denote the downloading time of the jth thread for request i by $X_{i,j}$, $j = 1, \cdots, n_i$. We assume that $X_{i,j}$ is *i.i.d.* exponentially distributed random variable with a mean μ due to the homogeneity assumption of all chunks.

We let T_A^i denote the arrival time of request i, and $T_S^{i,j}$ denote the starting time of the jth thread of request i. Without loss of generality, we assume $T_S^{i,1} \leq T_S^{i,2} \leq \cdots \leq T_S^{i,n_i}$. We denote the finishing time of the thread j of the request i as $T_F^{i,j}$, which is given by $T_F^{i,j} = T_S^{i,j} + X_{i,j}$. Note that the threads are ordered by their starting time but not the completion time. So it is possible that $T_F^{i,j} > T_F^{i,l}$ even if $j < l$. The departure time of request i, denoted as T_F^i, is then given by the time when k of its threads have finished. Let $T_F^{i,1:n_i} \leq T_F^{i,2:n_i} \leq \cdots \leq T_F^{i,n_i:n_i}$ be the sorted permutation of the finish times of request i's threads. Thus, we have $T_F^i = T_F^{i,k:n_i}$.

The queueing delay for request i, denoted as D_i is therefore given by

$$D_i = T_F^i - T_A^i. \tag{1}$$

Hence, the total expected delay under a thread allocation policy π is given by

$$\mathbb{E}_\pi[D] = \lim_{T \to \infty} \frac{1}{N_T} \sum_{i=1}^{N_T} \mathbb{E}_\pi[D_i], \tag{2}$$

where the expectation is with respect to the distributions of arrival process and departure process.

In the following, we will develop a new queueing model for analyzing such cloud systems that use FEC codes for data storage. Before we proceed, we first have the following definitions.

***Definition** 4.1:* The queue is said to be stable under a policy π if

$$\mathbb{E}_\pi[D] < \infty. \tag{3}$$

***Definition** 4.2: Capacity Region Λ:* The set of request arrival rates under which the queue can be stabilized by some possible scheme.

***Definition** 4.3: Throughput-optimal scheme:* A scheme is said to be throughput optimal if the queue with arbitrary arrival rate λ can be stabilized under this scheme whenever there exists an $\epsilon > 0$ such that $\lambda + \epsilon \in \Lambda$.

***Definition** 4.4: Delay-optimal scheme:* A scheme π is said to be delay optimal if it yields the smallest $\mathbb{E}_\pi[D]$ among all schemes for any request arrival rate λ such that $\lambda + \epsilon \in \Lambda$ for some $\epsilon > 0$.

B. Thread Allocation Schemes

In this section, we first present several thread allocation schemes that can be adopted by the dispatcher and then we motivate our problem using a simple example. The model can be depicted as a multiple-server queue with arrivals of data retrieving requests and a server corresponding to a thread. However, this is not a typical queueing model because a server can terminate its operation, depending on when other servers complete their service (due to the redundancy that is spread across the threads). We here consider two schemes: the greedy scheme and the sharing scheme.

1) The greedy scheme: All L threads are always allocated to the HoL (head-of-line) request simultaneously until it departs. During the serving process, if any thread out of L finishes downloading its assigned chunk, it immediately starts to download another distinct chunk belonging to the *same* file. Then, at some point, one thread finishes downloading so that the cumulative number of successfully downloaded chunks reaches k, all the other $L-1$ threads in progress get terminated immediately. The read request is considered complete and departs the queue. After that, all the threads will be allocated to serve the next HoL read request if the queue is not empty. Otherwise, all threads remain idle.

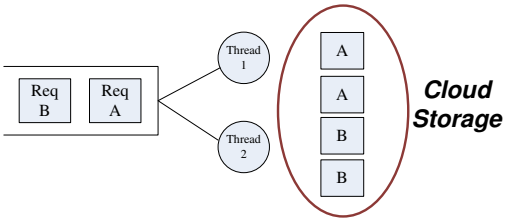

Fig. 5. An example of a two-thread queue

2) The sharing scheme: The dispatcher always allocates exactly k threads to each request (not necessarily simultaneously). The requests are served in a first-come-first-serve manner. The dispatcher allocates as many available threads as possible to a request until k threads have been assigned to it in total. When the number of allocated threads for a request reaches k, the dispatcher will allocate the available threads to the next request.

We can see that the greedy scheme and the sharing scheme are two extremes. The greedy scheme allocates the maximum possible resources to each individual request, but it can serve only one request at any time. On the other hand, the sharing scheme is the most conservative for each individual request, yet serves the maximum possible parallel requests. Another observation is that the total number of chunks required for the greedy scheme is $L + k - 1$, which consists of k successfully downloaded and $L - 1$ terminated. This corresponds to the assumption $n \geq L + k - 1$.

To better motivate our problem, we consider a simple example as shown in Fig. 5. Two threads are used to read two files A and B, which use coding parameters $(2, 1)$, i.e., each file simply has two duplications. We compare the delay of the two files using the two schemes under two different downloading time distributions.

Case 1) If the downloading time is constant for any chunk, we can see that the sharing scheme always outperforms the greedy scheme in terms of the delay performance since the parallel downloads do not bring any benefit when the delay is a fixed value.

Case 2) Consider a distribution of downloading time, which is 0 with probability 2/3 and 3000*ms* with probability 1/3. Therefore, we can see that sharing scheme has an expected delay of 1000*ms* for each request, since it allocates one request to each thread. On the other hand, since the greedy scheme allocates both threads to a request one by one, we can easily check that the expected delay for the first request is 1000/3 *ms* and the expected delay for the second one is 2000/3 *ms*. Both experience smaller delay than those under the sharing scheme.

The detailed calculation is as follows:

1) For the first request, it has a delay of 3000*ms* if and only if both threads suffer a delay of 3000*ms*. The probability of this scenario is 1/9. With a probability of 8/9, the delay is 0. Therefore, the expected delay for the first request is $\frac{3000}{9} + \frac{0\times 8}{9} = 1000/3$ *ms*.
2) For the second request, the expected delay is the sum of the expected waiting delay in the queue and its expected service delay. Notice that its expected waiting delay in the queue is exactly the expected delay of the first request, i.e., 1000/3 *ms*. In addition, its service delay can be calculated in the same way as the first request, that is, 1000/3 *ms*. Thus, the expected delay for the second request is 2000/3 *ms*.

Hence, we can observe that the delay-optimal scheme heavily depends on the distribution of the downloading time. This motivates us to ask a key question: *what is the delay optimal thread allocation scheme when the downloading time exhibits a distribution like the one in Fig. 3*. In the following sections, as we discussed in Section III, we approximate the downloading time for any individual thread as an *i.i.d.* exponentially distributed random variable with parameter μ.

V. Analysis of Case $k = 1$

To facilitate the understanding of our results, we start by analyzing the case when $k = 1$. In this case, it simply means using data duplication to store the files. To present our analysis, we first define the following:

***Definition** 5.1: Work-conserving schemes:* A scheme is said to be work-conserving if no thread is idle whenever there are requests waiting in the queue.

We emphasize that **a throughput-optimal scheme must be work-conserving**. In addition, **a delay-optimal scheme must be work-conserving**. The reason is that the delay and throughput performance of any non-work-conserving scheme can be improved by assigning the idle threads to download some additional chunks.

Before we state our results, a key property of work-conserving scheduling schemes for both $k = 1$ and $k > 1$ cases is described as follows:

***Lemma** 5.2:* If the downloading time of each individual thread is *i.i.d.* exponentially distributed with rate μ, and all L threads are active, then the service time for the threads to download one more coded chunk is exponentially distributed with rate $L\mu$ and is *i.i.d.* across coded chunks.

Proof: Suppose that the system starts to download the next coded chunk at time t, either because a new request arrives or because a coded chunk is downloaded at t. The downloading operation of Thread l may start before time t. Let R_l be the resident downloading time of Thread l after time t. Since all L threads are active, the service time for the threads to download one more coded chunk is given by

$$R = \min_{l\in\{1,\cdots,L\}} R_l. \tag{4}$$

According to the memoryless property of exponential distribution, the R_l's are also exponentially distributed with rate μ and are *i.i.d.* across threads. Therefore, $R = \min_{l\in\{1,\cdots,L\}} R_l$ is exponentially distributed with rate $L\mu$. By the memoryless property of exponential distribution, the download durations of the retrieved coded chunks are *i.i.d.* Therefore, the asserted statement is proved. ■

Now, for $k = 1$ case, we have the following result.

***Theorem** 5.3:* When $k = 1$ and $n \geq L$, given that the downloading time of each individual thread is *i.i.d.* exponentially distributed, any work-conserving scheme is throughput optimal and also delay optimal for any arrival process.

Proof: First, we show that a delay optimal scheme must also be throughput optimal. Suppose this is not true and denote a delay optimal scheme in consideration as π^*, which is not throughput optimal. Then, according to Definitions 4.1 and 4.3, there exists an arrival rate λ such that $\lambda + \epsilon \in \Lambda$, but under this rate, π^* results in $\mathbb{E}_{\pi^*}[D] = \infty$. However, since $\lambda + \epsilon \in \Lambda$, there exists a policy π that has $\mathbb{E}_{\pi}[D] < \infty$, and hence $\mathbb{E}_{\pi}[D] < \mathbb{E}_{\pi^*}[D]$. This contradicts the delay optimality of π^*.

Now we prove the delay optimality part. We only need to consider the work-conserving schemes, because a non-work-conserving scheme cannot be delay optimal. In particular, we can always transform a non-work-conserving scheme to a work-conserving scheme by utilizing the idle threads to download some additional chunks, which leads to a lower delay.

Since $k = 1$, each downloaded coded chunk leads to a request departure. According to Lemma 5.2, the service durations of the downloaded coded chunks are *i.i.d.* exponentially distributed under any work-conserving scheme. Therefore, the service durations of the requests are also *i.i.d.* exponentially distributed. Different work-conserving schemes only affect the service order of different requests, but have no influence on the average delay. Therefore, any work-conserving scheme is delay optimal. By this, the asserted statement is proved.

■

Remark: It is interesting and somewhat surprising to see that although some work-conserving schemes, such as the greedy scheme, appears to "waste" some system resources because some threads have useless (unfinished) downloads due to thread terminations, it is still both throughput and delay optimal. The reason is because the abandoned threads are the ones suffering long delay. The service time for any particular file is reduced compared to the non-wasted schemes. We will still observe such "wastage but optimality" in the next section.

VI. ANALYSIS OF CASE $k > 1$

In this section, we extend our results to the case when $k > 1$. We first have the following definitions.

***Definition** 6.1: Effective chunks:* chunks that are downloaded by the first k completed threads for any particular file.

***Definition** 6.2: Thread terminations:* chunk download attempts that are terminated due to the completion of the read request, i.e., there are already k effective chunks downloaded.

Next, we have the following theorem regarding the throughput optimality of the work-conserving scheduling policies.

***Theorem** 6.3:* When $k > 1$ and $n \geq L+k-1$, given that the downloading time of each individual thread is *i.i.d.* exponentially distributed, any work-conserving scheme is throughput optimal.

Proof: We only need to consider the work-conserving schemes, because a non-work-conserving scheme cannot be throughput optimal. According to Lemma 5.2, we know that the service durations of effective chunks under any work-conserving scheme have the same distribution, i.e., they are *i.i.d.* exponentially distributed with rate $L\mu$. Since each request requires to download exactly k effective chunks, the average request departure rate of any work-conserving scheme is $L\mu/k$. On the other hand, the queue is stable if and only if the average request arrival rate is less than the average request departure rate. From the previous discussion, any work-conserving scheme can provide the maximum request departure rate $L\mu/k$. Therefore, any work-conserving scheme is throughput optimal.

■

We now have the following theorem regarding the delay performance of the greedy scheme.

***Theorem** 6.4:* When $k > 1$ and $n \geq L + k - 1$, given that the downloading time of each individual thread is *i.i.d.* exponentially distributed, the greedy scheme is delay optimal for any arrival process.

Proof: First, notice that a delay-optimal scheme must be work-conserving. Otherwise, it is easy to reduce the delay by simply allocating the idle threads to download more chunks.

Let s_i denote the arrival instants of the i-th arrival effective chunk and t_i denote the departure instants of the i-th departed effective chunk. Obviously, we have $s_i < s_{i+1}$, $t_i < t_{i+1}$, and $s_i < t_i$. Fix the arrival process $\omega_A = \{s_1, s_2, \cdots\}$ of the effective chunks, we will show that the distribution (probability density function) of the departure process $\omega_D = \{t_1, t_2, \cdots\}$ remains the same for any work-conserving scheme.

According to Lemma 5.2, the service time of an effective chunk has the same distribution under any work-conserving scheme. Let S_i denote the service time of the i-th departed effective chunk. Given the arrival instant s_1 of the first effective chunk, the departure instant t_1 is given by $t_1 = s_1 + S_1$. Moreover, since the distribution of S_i remains the same under any work-conserving scheme, the distribution of t_1 also remains the same under any work-conserving scheme. Now consider the departure instant t_i of the i-th departed effective chunk. Since the system is work-conserving, t_i is given by $t_i = \max\{s_i, t_{i-1}\} + S_i$. Since the distributions of $\{t_1, t_2, \cdots, t_{i-1}\}$ and S_i remain unchanged under any work-conserving scheme, one can show that the distribution of $\{t_1, t_2, \cdots, t_i\}$ also remains unchanged under any work-conserving scheme. By induction, we attained that the distribution of the departure process $\omega_D = \{t_1, t_2, \cdots\}$ remains the same under any work-conserving scheme.

Next, we will show that the greedy scheme has the smallest delay among all work-conserving schemes. According to Eqns.

(1) and (2), the expected delay is given by:

$$\begin{aligned}
&\mathbb{E}_\pi[D] \\
&= \lim_{T\to\infty} \frac{1}{N_T} \sum_{i=1}^{N_T} \mathbb{E}_\pi[T_F^i - T_A^i] \\
&= \lim_{T\to\infty} \frac{1}{N_T} \sum_{i=1}^{N_T} \int_{\omega_A} \int_{\omega_D} (T_{F:\pi}^i - T_A^i) f(\omega_A) f(\omega_D|\omega_A) d\omega_A d\omega_D, \\
&= \lim_{T\to\infty} \int_{\omega_A} \int_{\omega_D} \frac{1}{N_T} \sum_{i=1}^{N_T} (T_{F:\pi}^i - T_A^i) f(\omega_A) f(\omega_D|\omega_A) d\omega_A d\omega_D,
\end{aligned} \tag{5}$$

where the expectation is taken over the distribution of the arrival process of effective chunks ω_A and the corresponding distribution of the departure process of the effective chunks ω_D. $T_{F:\pi}^i$ denotes the departure time of the $i-$th request under scheme π. Notice that the distribution of ω_A and ω_D are always the same for any work-conserving scheme.

For the greedy scheme π_{greedy}, since it serves the requests one by one and each request requires exactly k effective chunks, the i-th request departs when the ik-th effective chunk departs, i.e.,

$$T_{F:\pi_{\mathsf{greedy}}}^i = t_{ik}. \tag{6}$$

For an alternative work-conserving scheme π_{alt}, suppose that the i-th arrival request departs when the a_i-th effective chunk departs, i.e.,

$$T_{F:\pi_{\mathsf{alt}}}^i = t_{a_i}. \tag{7}$$

The sequence $\{t_{a_1}, t_{a_2}, \cdots, t_{a_{N_T}}\}$ may not be in an increasing order. After sorting, suppose that this sequence becomes $\{t_{b_1}, t_{b_2}, \cdots, t_{b_{N_T}}\}$ such that $t_{b_i} < t_{b_{i+1}}$ and $t_{b_i} \in \{t_{a_1}, t_{a_2}, \cdots, t_{a_{N_T}}\}$. Therefore, i requests have departed from the system by time t_{b_i}. Since each request contains k effective chunks, the systems must have downloaded at least ik effective chunks by t_{b_i}, which tells us that

$$t_{b_i} \geq t_{ik}. \tag{8}$$

Using (6)-(8), we can attain

$$\begin{aligned}
&\frac{1}{N_T} \sum_{i=1}^{N_T} (T_{F:\pi_{\mathsf{greedy}}}^i - T_A^i) - \frac{1}{N_T} \sum_{i=1}^{N_T} (T_{F:\pi_{\mathsf{alt}}}^i - T_A^i) \\
=&\frac{1}{N_T} \sum_{i=1}^{N_T} (T_{F:\pi_{\mathsf{greedy}}}^i - T_{F:\pi_{\mathsf{alt}}}^i) \\
=&\frac{1}{N_T} \sum_{i=1}^{N_T} (t_{ik} - t_{a_i}) \\
=&\frac{1}{N_T} \sum_{i=1}^{N_T} (t_{ik} - t_{b_i}) \\
\leq&0.
\end{aligned} \tag{9}$$

Since the distribution of ω_A and ω_D are always the same for any work-conserving scheme, (9) tells us that the greedy scheme achieves the minimum average delay among all the work-conserving schemes, which completes our proof. ■

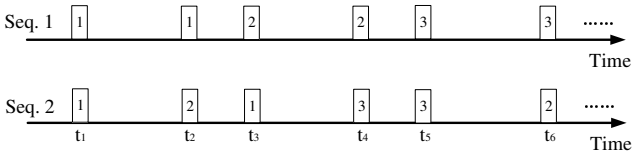

Fig. 6. Different departure sequences under same sample path

Fig. 6 shows an example when $k = 2$, i.e., each request requires two effective chunks. In the figure, we plot two same sample paths of effective chunk departure, where each rectangle represents an effective chunk and the number means which request this effective chunk belongs to. Sequence 1 represents the effective chunks belonging under the greedy scheme, where the chunks depart in order. On the contrary, some other work-conserving scheme may have a non-ordered effective chunks departure, such as Sequence 2. Although for a particular request, its delay may be greater in greedy scheme, such as request 3 departs later in Sequence 1 than Sequence 2 in the figure, the summation of delay under greedy scheme is always no greater than any other work-conserving scheme, as $t_2 + t_4 + t_6 \leq t_3 + t_5 + t_6$ in the example.

A. Discussion

Why "wastage but optimality"?: Similarly to the case where $k = 1$, our result may look counter-intuitive at first. It seems that the greedy policy allocates more-than-necessary threads to each file. Because some of the threads are redundant, they will be terminated upon the completion of the first k chunks. How can a scheme with such "wastage" be optimal? The key insight here is that such redundancy allows us to "cut the tail", i.e., the threads with relatively long downloading time get terminated. Furthermore, because the tail is shorter, the total consumed resource, defined as the number of threads multiplies the average length of thread occupation, does not increase (under exponential service time). In other words, the redundancy is not wasted because it contributes to the exploration of the shorter tail and thus reduces the amount of resource that would have been consumed by the long tail.

VII. Simulations

In this section, we conduct experiments under exponentially distributed downloading time and real traces plotted in Section III.

A. Simulation Setup

We simulate a system with $L = 16$ threads. The downloading requests arrive as a Poisson process with parameter λ. In our simulations, we set $\lambda = 50$, and the number of arrival is set to be 62500. The expected downloading time for any individual thread is assumed to be $1/\mu$. Thus, the load is given by

$$\rho = \frac{\lambda}{L\mu}. \tag{10}$$

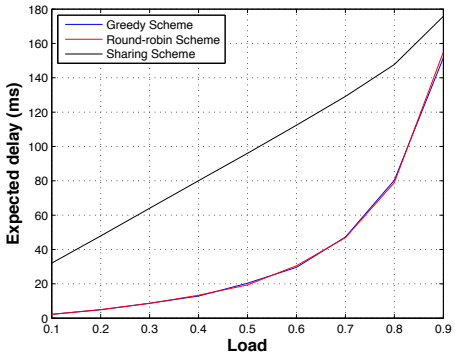

Fig. 7. Expected delay under greedy, round-robin and sharing schemes when $k = 1$

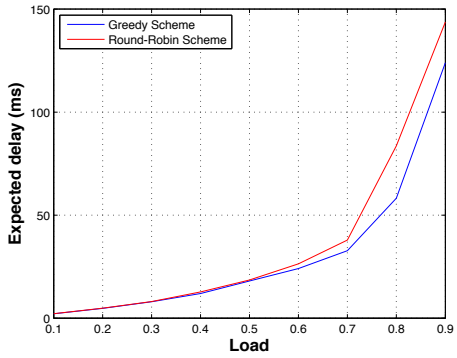

Fig. 8. Expected delay under greedy and round-robin schemes when $k = 2$

Besides the greedy scheme and the sharing scheme, we will also simulate the round-robin scheme as follows:

The Round-robin scheme: At any time when some threads become idle and the queue is not empty, the dispatcher allocates these idle threads to all the requests in the queue in a round-robin way, i.e., the first idle thread to the first request, the second idle request to the second request, etc..

Notice that the round-robin scheme is also work-conserving. De facto, the Round-robin scheme actually works closely to the greedy for low to medium arrival rates as the probability of finding two requests waiting in the queue is low. We take the average delay over 1000 sample paths for each experiment.

B. Exponential Distribution Case

First, we show the result under the case of $k = 1$. We set $n = L+k-1$. We plot the expected delay versus different load ρ under the greedy scheme, the round-robin scheme and the sharing scheme. As shown in Fig. 7, we can see that the delay performance of the greedy scheme and round-robin scheme are the same, which is much smaller than the delay of the sharing scheme. The reason is that both of the greedy and round-robin scheme are work-conserving, while the sharing scheme is not. This observation validates our result in Theorem 5.3.

Fig. 8 shows the expected delay under both the greedy and the round-robin scheme when $k = 2$ (Since the sharing scheme has a much worse expected delay, in order to clearly show the gap between the two work-conserving schemes, we do not plot the sharing scheme in the figure). We can see that the greedy scheme outperforms the round-robin scheme in terms of expected delay, which verifies our result in Theorem 6.4. It is indeed surprising that the greedy scheme outperforms the Round-robin scheme as one can intuitively think that round robin makes more opportunistic use of parallel threads. However, the capacity regions are the same under both schemes, as we have proven in Theorem 6.3.

In Fig. 9, we fix $\rho = 0.1$. It shows that expected delay versus different values of k. It can be also noticed that the expected delay under the greedy scheme is strictly smaller than the expected delay under the round-robin scheme. And

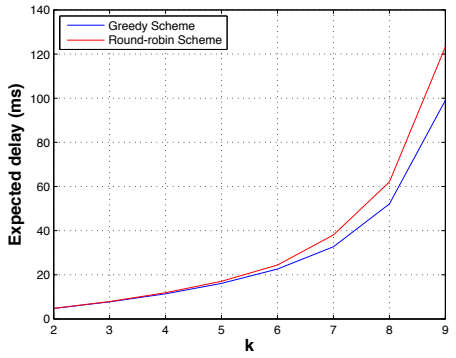

Fig. 9. Expected delay versus k under greedy and round-robin schemes

the gap increases as k becomes larger. The reason is that when k increases, the effective chunk departure sequence under round-robin scheme has more randomness and becomes more disordered. From the discussion following Theorem 6.4, we know that the ordered departure of requests under the greedy scheme leads to the smallest delay. Thus, a larger delay is expected if more disorder occurs.

C. Real Traces

We here adopt the real traces of downloading delay that are plotted in Section III.

In Fig. 10, we plot the read delay spread under different level of parallel downloading when $k = 1$. The red curve in the figure represents the original CCDF, that is, without making use of FEC redundancy. The other four curves represent the CCDF of read delay using different number of parallel threads to retrieve the file, which are 2, 4, 8, and 16, respectively. We can see that the delay spread can be significantly reduced by leveraging the FEC redundancy. And as the level of parallelism increases, the delay spread becomes narrower, resulting in a smaller latency.

VIII. Conclusion

In this paper, we study delay performance of downloading data from cloud storages by leveraging multiple parallel threads,

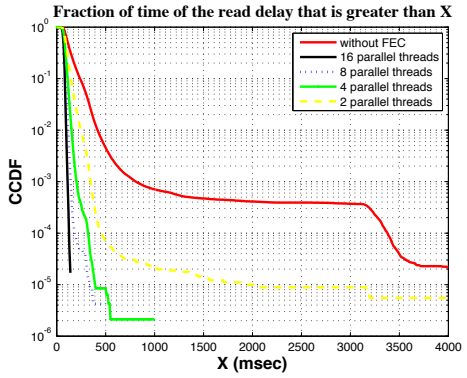

Fig. 10. CCDF of read delay under different level of parallel downloading

assuming that the data in the clouds has been encoded using FEC codes. This leads to a new queueing model. According to our observations on real traces that are measured over Amazon S3, we approximate the downloading time for any individual thread as an *i.i.d.* exponentially distributed random variable. We show that the any work-conserving scheme is delay-optimal when $k = 1$. When $k > 1$, it appears surprising that a simple greedy scheme is delay optimal. We validate our results through simulations with exponentially distributed service time. By using real traces, we show that making use of inherent redundancy introduced by FEC can indeed reduce the delay and significantly cut the delay spread.

REFERENCES

[1] "Public/private cloud storage market," http://www.marketsandmarkets.com/PressReleases/cloud-storage.asp.

[2] "Wuala," http://www.wuala.com/.

[3] G. Liang and U. Kozat, "FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage with Coding," *IEEE/ACM Trans. Networking*, Nov 2013, preprint.

[4] S. L. Garfinkel, "An evaluation of Amazons grid computing services: EC2, S3 and SQS," Harvard University, Tech. Rep., 2007.

[5] Y. Lu, Q. Xie, G. Kliot, A. Geller, J. Larus, and A. Greenberg, "Join-idle-queue: A novel load balancing algorithm for dynamically scalable web services," *Performance Evaluation*, 2011.

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory, vol. 56, no. 9*, Sept. 2010.

[7] Y. Wu and A. G. Dimakis, "Reducing repair traffic for erasure coding-based storage via interference alignment," *IEEE Intl Symp. on Information Theory (ISIT)*, 2009.

[8] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran, "Explicit and optimal exact-regenerating codes for the minimum-bandwidth point in distributed storage," *IEEE Intl Symp. on Information Theory (ISIT)*, 2010.

[9] D. Leong, A. G. Dimakis, and T. Ho, "Distributed storage allocation problems," *Proceedings of the Workshop on Network Coding, Theory, and Applications (NetCod)*, 2009.

[10] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A survey on network codes for distributed storage," *Proceedings of the IEEE, vol. 99, no. 3*, March 2011.

[11] T. Dikaliotis, A. G. Dimakis, T. Ho, and M. Effros, "On the delay of network coding over line networks," *IEEE International Symposium on Information Theory (ISIT)*, 2009.

[12] U. J. Ferner, M. Medard, and E. Soljanin, "Toward sustainable networking: Storage area networks with network coding," http://arxiv.org/abs/1205.3797, 2012.

[13] A. ParandehGheibi, M. Medard, A. Ozdaglar, and S. Shakkottai, "Avoiding interruptions: A QoE reliability function for streaming media applications," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 5, pp. 1064–1074, May 2011.

[14] J. H. Kim, H.-S. Ahn, and R. Righter, "Managing queues with heterogeneous servers," *Journal of Applied Probability*, vol. 48, pp. 435–452, 2011.

[15] Y. Kim, R. Righter, and R. Wolff, "Grid scheduling with NBU service times," *Operations Research Letters*, vol. 38, pp. 502–504, 2010.

[16] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes can reduce queueing delay in data centers," *Proceedings of IEEE International Symposium on Information Theory (ISIT)*, July 2012.

[17] N. B. Shah, K. Lee, and K. Ramchandran, "The MDS queue: Analysing the latency performance of erasure codes," http://arxiv.org/abs/1211.5405, 2013.

[18] G. Joshi, Y. Liu, and E. Soljanin, "On the delay-storage trade-off in content download from coded distributed storage systems," http://arxiv.org/abs/1305.3945, 2013.