# TOFEC: Achieving Optimal Throughput-Delay Trade-off of Cloud Storage Using Erasure Codes

Guanfeng Liang and Ulaş C. Kozat
DOCOMO Innovations, Inc., Palo Alto, CA 94304
Email: {gliang,kozat}@docomoinnovations.com

***Abstract*—Our paper presents solutions using erasure coding, parallel connections to storage cloud and limited chunking (i.e., dividing the object into a few smaller segments) together to significantly improve the delay performance of uploading and downloading data in and out of cloud storage.**

**TOFEC is a strategy that helps front-end proxy adapt to level of workload by treating scalable cloud storage (e.g. Amazon S3) as a shared resource requiring admission control. Under light workloads, TOFEC creates more smaller chunks and uses more parallel connections per file, minimizing service delay. Under heavy workloads, TOFEC automatically reduces the level of chunking (fewer chunks with increased size) and uses fewer parallel connections to reduce overhead, resulting in higher throughput and preventing queueing delay. Our trace-driven simulation results show that TOFEC's adaptation mechanism converges to an appropriate code that provides the optimal delay-throughput trade-off without reducing system capacity. Compared to a non-adaptive strategy optimized for throughput, TOFEC delivers $2.5\times$ lower latency under light workloads; compared to a non-adaptive strategy optimized for latency, TOFEC can scale to support over $3\times$ as many requests.**

***Index Terms*—FEC, Cloud storage, Queueing, Delay**

## I. Introduction

Cloud storage has been gaining popularity rapidly as an economic, flexible and reliable data storage service that many cloud-based applications nowadays are implemented on. Typical cloud storage systems are implemented as key-value stores in which data objects are stored and retrieved via their unique keys. To provide high degree of availability, scalability, and data durability, each object is replicated several times within the internal distributed file system and sometimes also further protected by erasure codes to more efficiently use the storage capacity while attaining very high durability guarantees [1].

Cloud storage providers usually implement a variety of optimization mechanisms such as load balancing and caching/prefetching internally to improve performance. Despite all such efforts, still evaluations of large scale systems indicate that there is a high degree of randomness in delay performance [2]. Thus, services that require more robust and predictable Quality of Service (QoS) must deploy their own external solutions such as sending multiple/redundant requests (in parallel or sequentially), chunking large objects into smaller ones and read/write each chunk through parallel connections, replicate the same object using multiple distinct keys, etc.

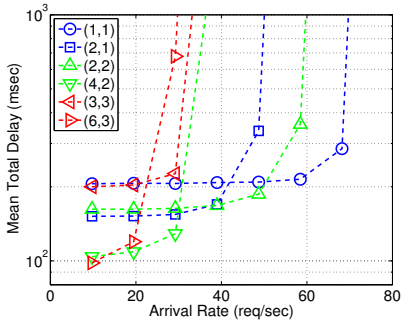



Fig. 1. Delay for downloading 3MB files using fixed MDS codes

In this paper, we present TOFEC – an **external** strategy that can provide much better throughput-delay performance for file accessing on cloud storage utilizing erasure coding that does not require any modification nor knowledge of the internal implementation of storage cloud. Although we base our analysis and evaluation on Amazon S3 service and present TOFEC as an external solution, TOFEC can be applied to many other cloud storage systems both externally and internally with small modifications. The latter can be accomplished, for example, by making TOFEC a thin layer on top of the original API.

### A. State of the Art

Among the vast amount of research on improving cloud storage system's delay performance emerged in the past few years, two groups in particular are closely related to our work presented in this paper:

**Erasure Coding with Redundant Requests:** As proposed by authors of [3], [4], [5], files are divided into a *predetermined* number of $k$ chunks, each of which is $1/k$ the size of the original file, and encoded into $n > k$ of "coded chunks" using an $(n, k)$ Maximum Distance Separable (MDS) code, or more generally a Forward Error Correction (FEC) code. Downloading/uploading of the original file is accomplished by downloading/uploading $n$ coded chunks using parallel connections simultaneously and is deemed served when download/upload of any $k$ coded chunks complete. Such mechanisms significantly improves the delay performance under light workload. However, as shown in our previous work [3] and later reconfirmed by [5], system capacity is reduced due to the overhead for using smaller chunks and redundant requests. This phenomenon is illustrated in Fig.1 where we plot the delay-throughput trade-off for using different MDS codes from

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

our simulations using delays traces collected on Amazon S3. Codes with different $k$ are grouped in different colors. Using a code with high level of chunking and redundancy, in this case a $(6, 3)$ code, although delivers $2\times$ gain in delay at light workload, reduces system capacity to only $30\%$ of the original basic strategy without chunking and redundancy, i.e., $(1, 1)$ code!

This problem is partially addressed in [3] where we present strategies that adjust $n$ according to workload level so that it achieves the near-optimal throughput-delay trade-off for the *predetermined* $k$. For example, if $k = 3$ is used, the strategies in [3] will achieve the lower-envelop of the red curves in Fig.1. Yet, it still suffers from an almost 60% loss in system capacity.

**Dynamic Job Sizing:** It has been observed in [2], [6] that in key-value storage systems such as Amazon S3 and Microsoft's Azure Storage, throughput is dramatically higher when they receive a small number of storage access requests for large jobs (or objects) than if they receive a large number of requests for small jobs (or objects), because each storage request incurs overheads such as networking delay, protocol-processing, lock acquisitions, transaction log commits, etc. Authors of [6] developed Stout in which requests are dynamically batched to improve delay-throughput trade-off of key-value storage systems. Based on the observed congestion Stout increase or reduce the batching size. Thus, at high congestion, a larger batch size is used to improve the throughput while at low congestion a smaller batch size is adopted to reduce the delay.

### B. Main Contribution

We introduce an adaptive strategy for accessing cloud storage systems via erasure coding, called TOFEC (Throughput Optimal FEC Cloud), that implements dynamic adjustment of chunking and redundancy levels to provide the optimal delay-throughput trade-off. In other words, TOFEC achieves the lower envelop of curves in all colors in Fig.1.

The primary novelty of TOFEC is its backlog-based adaptive algorithm for dynamically adjusting the chunk size as well as the number of redundant requests issued to fulfill storage access requests. This algorithm of variable chunk sizing can be viewed as a novel integration of prior observations from the two bodies of works discussed above. Based on the observed backlog level as an indicator of the workload, TOFEC increases or reduces the chunk size, as well as the number of redundant requests. In our trace-driven simulation evaluation, we demonstrate that: (1) TOFEC successfully adapts to full range of workloads, delivering $3\times$ lower average delay than the basic static strategy without chunking under light workloads, and under heavy workloads over $3\times$ the throughput of a static strategy with high chunking and redundancy levels optimized for service delay; and (2) TOFEC provides good QoS guarantees as it delivers low delay variations.

TOFEC works without any explicit information from the back-end cloud storage implementation: its adaptation strategy is implemented solely at the front-end application server (the storage client) and is based exclusively on the measured latency from unmodified cloud storage systems. This allows TOFEC to be more easily deployed, as individual cloud applications can adopt TOFEC without being tied-up with any particular cloud storage system, as long as a small number of APIs are provided by the storage system.

## II. System Models

### A. Basic Architecture and Functionality

The basic system architecture of TOFEC captures how web services today utilize public or private storage clouds. The architecture consists of proxy servers in the front-end and a key-value store, referred to as storage cloud, in the back-end. Users interact with the proxy through a high-level API and/or user interfaces. The proxy translates every high-level user request (to read or write a file) into a set of $n \geq 1$ tasks. Each task is essentially a basic storage access operation such as `put, get, delete,` etc. that will be accomplished using low-level APIs provided by the storage cloud. The proxy maintains a certain number of parallel connections to the storage cloud and each task is executed over one of these connections. After a certain number of tasks are completed successfully, the user request is considered accomplished and the proxy responds to the user with an acknowledgment. The solutions we present are deployed on the proxy server side transparent to the storage cloud.

For read request, we assume the file is pre-coded into $n^{max} \geq n$ coded chunks with an $(n^{max}, k)$ MDS code and stored on the cloud. Completion of downloading any $k$ coded chunks provides sufficient data to reconstruct the requested file. For write request, the file to be uploaded is divided and encoded into $n$ coded chunks using an $(n, k)$ MDS code and hence completion of uploading any $k$ coded chunks means sufficient data have been stored onto the cloud. Thus, upon completion of a request, the $n-k$ un-started and/or unfinished tasks are then preemptively canceled and removed from the system.[1]

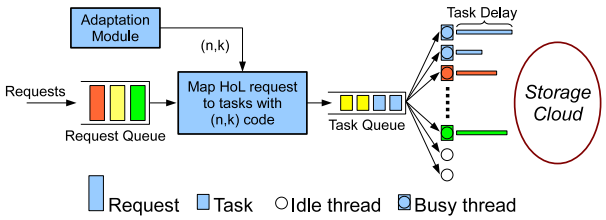



Fig. 2. System Model

Accordingly, we model the proxy by the queueing system shown in Fig.2. There are two FIFO (first-in-first-out) queues: (i) the *request queue* that buffers all incoming user requests, and (ii) the *task queue* that is a multi-server queue and holds all tasks waiting to being executed. $L$ threads[2], representing the set of parallel connections to the storage cloud, are attached to the task queue. The adaptation module of TOFEC monitors the state of the queues and the threads, and decides what

[1]For write request, the remaining tasks can also be scheduled as background jobs depending on the subsequent read profile of the file.

[2]We avoid the term "server" that is commonly used in queueing theory literature to prevent confusion.

coding parameter $(n, k)$ to be used for each request. Without loss of generality, we assume that the head-of-line (HoL) request leave the request queue only when there is at least one idle thread **and** the task queue is empty. A batch of $n$ tasks are then created for that request and injected into the task queue. As soon as any $k$ tasks complete successfully, the request is considered completed. Such a queue system is work conserving since no thread is left idle as long as there is any request or task pending.

### B. Basics of Erasure Codes

An $(n, k)$ MDS code (e.g., Reed-Soloman codes) encodes $k$ data chunks each of $B$ bits into a codeword consisting of $n$ $B$-bit long coded chunks. The coded chunks can sustain up to $n - k$ erasures such that the $k$ original data chunks can be efficiently reconstructed from **any** subset of $k$ coded chunks. $n$ and $k$ are called the length and dimension of the MDS code. We also define $r = n/k$ as the redundancy ratio of an $(n, k)$ MDS code. This erasure resistant property of MDS codes has been utilized in prior works [3], [4], [5], as well as in this paper, to improve delay of cloud storage systems. Essentially a coded chunk experiencing long delay is treated as an erasure.

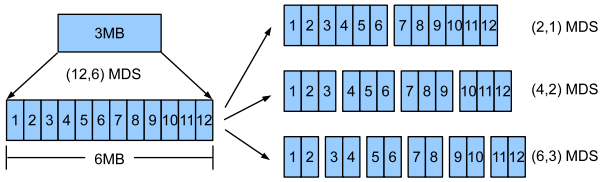



Fig. 3. Example of supporting multiple chunk sizes with Shared Key approach: the 3MB file is divided and encoded into a coded file of 6MB consisting 12 strips, each of 0.5MB. Download the file using a $(2, 1)$ MDS code is accomplished by creating two read tasks: one for strips 1-6, and the other for strips 7-12.

In this paper, we make use of another interesting property of MDS codes to implement variable chunk sizing of TOFEC in a storage efficient manner: MDS code of high length and dimension for small chunk size can be used as MDS code of smaller code length and dimension for larger chunk size. To be more specific, consider any $(N, K)$ MDS code for chunks of $b$ bits. To avoid confusion, we will refer to these $b$-bit chunks as strips. A different MDS code of length $n = N/m$, dimension $k = K/m$ and chunk size $B = bm$ for some $m > 1$ can be constructed by simply batching every $m$ data/coded strips into one data/coded chunk. The resulting code is an $(n, k)$ MDS code for $B$-bit chunks because any $k$ coded chunks covers $mk = K$ coded strips, which is sufficient to reconstruct the original file of $Bk = bm \times K/m = bK$ bits. This property is illustrated as an example in Fig. 3. In this example, a 3MB file is divided into 6 strips of 0.5MB and encoded into 12 coded strips of total size 6MB, using a $(12, 6)$ MDS code. This code can then be used as a $(2, 1)$ code for 3MB chunks, a $(4, 2)$ code for 1.5MB chunks and a $(6, 3)$ code for 1MB chunks **simultaneously** by batching 6, 3 and 2 strips into a chunk.

### C. Definitions of Different Delays

The delay experienced by a user request consists of two components: *queueing delay ($D_q$)* and *service delay ($D_s$)*. Both are defined with respect to the request queue: (i) the queueing delay is the amount of time a request spends waiting in the request queue and (ii) the service delay is the period of time between when the request leaves the request queue (i.e., admitted into the task queue and started being served by at least one thread) and when it finally leaves the system (i.e., the first time when any $k$ of the corresponding tasks complete). In addition, we also consider the *task delays ($D_t$)*, which is the time it takes for a thread to serve a task assuming it is not terminated or canceled preemptively. To clarify these definitions of delays, consider a request served with an $(n, k)$ MDS code, with $T_A$ its arrival time, $T_1 \leq T_2 \leq \cdots \leq T_n$ the starting times of the corresponding $n$ tasks[3]. Then the queueing delay is $D_q = T_1 - T_A$. Suppose $D_{t,1}, \cdots, D_{t,n}$ are the corresponding task delays, then the completion times of these task will be $X = \{T_1 + D_{t,1}, \cdots, T_n + D_{t,n}\}$ if none is canceled. So the request will leave the system at time $X_{(k)}$, which denotes the $k$-th smallest value in $X$, i.e., the time when $k$ tasks complete. Then the service delay of this request is $D_s = X_{(k)} - T_1$.

## III. Variable Chunk Sizing

In this section, we discuss implementation issues as well as pros and cons of two potential approaches, namely *Unique Key* and *Shared Key*, for supporting erasure-code-based access to files on the storage cloud with a variety of chunk sizes. Suppose the maximum desired redundancy ratio is $r$, then these approaches implement variable chunk sizing as follows:

- **Unique Key:** For every choice of chunk size (or equivalently $k$), a separate batch of $rk$ coded chunks are created and each coded chunk is stored as an individual object with its unique key on the storage cloud. The access to different chunks is implemented through basic `get`, `put` storage cloud APIs.
- **Shared Key:** A coded file is first obtained by stacking together the coded strips obtained by applying a high-dimension $(N = rK, K)$ MDS code to the original file, as described in Section II-B and illustrated in Fig.3. For read, the coded file is stored on the cloud as one object. Access to chunks with variable size is realized by downloading segments in the coded file corresponding to batches of a corresponding number of strips, using a same key with more advanced "partial read" storage cloud APIs. Similarly, for write, the file is uploaded in parts using "partial write" APIs and then later merged into one object in the cloud.

### A. Implementation and Comparison of the two Approaches

*1) Storage cost:* When the user request is to write a file, storage cost of Unique Key and Shared Key is not so different. However, to support variable chunk sizing for read requests, Shared Key is significantly more cost-efficient than Unique Key. With Shared Key, a single coded file stored on the cloud can be reused to support essentially an arbitrary number of

[3]We assume $T_i = \infty$ if the $i$-th task is never started.

different chunk sizes, as long as the strip size is small enough. On the other hand, it seems impossible to achieve similar reusing with the Unique Key approach where different chunks of the same file is treated as individual objects. So with Unique Key, every additional chunk size to be supported requires an extra storage cost $r\times$ file size. Such linear growth of storage cost easily makes it prohibitively expensive even to support a small number of chunk sizes.

*2) Diversity in delays:* The success of TOFEC and other proposals to use redundant requests (either with erasure coding or replication) for delay improvement relies on diversity in cloud storage access delays. In particular, TOFEC, as well as [3], [4], [5], requires access delays for different chunks of **the same file** to be weakly correlated.

With Unique Key, since different chunks are treated as individual objects, there is no inherent connection among them from the storage cloud system's perspective. So depending on the internal implementation of object placement policy of the storage cloud system, chunks of a file can be stored on the cloud in different storage units (disks or servers) on the same rack, or in different racks in the same data center, or even to different data centers at distant geographical locations. Hence it is quite likely that delays for accessing different chunks of the same file show very weak correlation.

On the other hand, with Shared Key, since coded chunks are combined into one coded file and stored as one object in the cloud, it is very likely that the whole coded file, hence all coded chunks/strips, is stored in the same storage unit, unless the storage cloud system internally divides the coded file into pieces and distributes them to different units. Although many distributed storage systems do divide files into parts and store them separately, it is normally only for larger files. For example, the popular Hadoop distributed file system by default does not divide files smaller than 64MB. When different chunks are stored on the same storage unit, we can expect higher correlation in their access delays. It then is to be verified that the correlation between different chunks with the Shared Key approach is still weak enough for our coding solution to be beneficial.

*3) Universal support:* Unique Key is the approach adopted in our previous work [3] to support erasure-code based file accessing with **one predetermined** chunk size. A benefit of Unique Key is that it only requires basic `get` and `put` APIs that all storage cloud systems must provide. So it is readily supported by all storage cloud systems and can be implemented on top of any one.

On the other hand, Shared Key requires more advanced APIs that allow the proxy to download or upload only the targeted segment of an object. Such advanced APIs are not currently supported by all storage cloud systems. For example, to the best of our knowledge currently Microsoft's Azure Storage provides only methods for "partial read"[4] but none for "partial write". On the contrary, Amazon S3 provides partial access for both read and write: the proxy can download a specific inclusive byte range within an object stored on S3 by calling `getObject(request,destination)`[5]; and for uploading an `uploadPart` method to upload segments of an object and an `completeMultipartUpload` method to merge the uploaded segments are provided. We expect more service providers to introduce both partial read and write APIs in the near future.

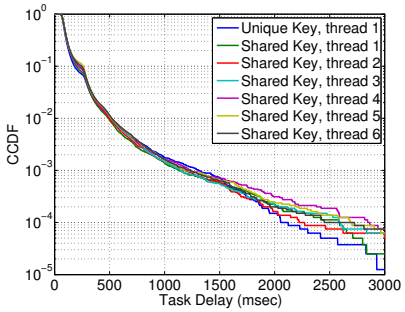



Fig. 4. CCDF of individual threads with 1MB chunks and $n = 6$

### B. Measurements on Amazon S3

To understand the trade-off between Unique Key and Shared Key, we run measurements over Amazon EC2 and S3. EC2 instance served as the proxy in our system model. We instantiated an extra large EC2 instance with high I/O capability in the same availability region as the S3 bucket that stores our objects. We conducted experiments on different week days in May to July 2013 with various chunk sizes between 0.5MB to 3MB and up to $n = 12$ coded chunks per file. For each value of $n$, we allow $L = n$ simultaneously active threads while the $i$-th thread being responsible for downloading the $i$-th coded chunk of each file. Each experiment lasted longer than 24 hours. We alternated between different settings to capture similar time of day characteristics across all settings.

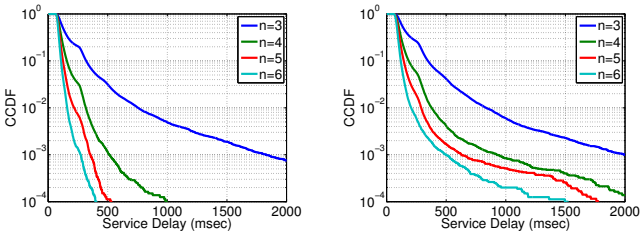
The experiments are conducted within all 8 availability regions in Amazon S3. Except for the "US Standard" availability region, all other 7 regions demonstrate similar performance statistics that are consistent over different times and days. We conjecture the different and inconsistent behavior of "US Standard" might be due to the fact that it targets a slightly different usage pattern and it may employ a different implementation for that reason[6]. We will exclude "US Standard" from subsequent discussions. Due to lack of space, we only show a limited subset of findings for availability region "North California" that are representative for regions other than "US Standard":

(1) In both Unique Key and Shared Key, the task delay distribution observed by different threads are almost identical. The two approaches are indistinguishable even beyond 99.9th percentile. Fig.4 show the complementary cumulative distribution function (CCDF) of task delays observed by individual threads for 1MB chunks and $n = 6$. Both approaches demonstrate large delay spread in all regions.

[4]E.g. `DownloadRangeToStream(target, offset, length)` downloads a segment of `length` bytes starting from the `offset`-th byte of the `target` object (or "blob" in Azure's jargon).

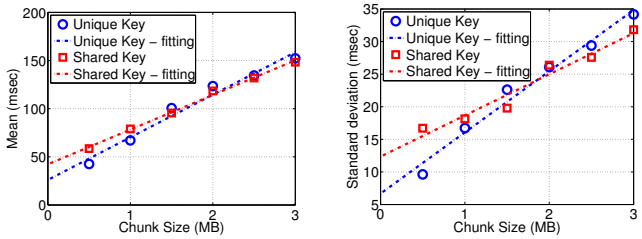
[5]The byte range is set by calling `request.setRange(start,end)`.

[6]See http://docs.aws.amazon.com/general/latest/gr/rande.html#s3_region




(a) Unique Key (b) Shared Key

Fig. 5. CCDF of service delay for reading 3MB files with 1MB chunks




(a) Mean (b) Standard Deviation

Fig. 6. Delay statistics vs. chunk size

(2) Task delays for different threads in Unique Key show close to zero correlation, while they demonstrate slightly higher correlation in Shared Key, as it is expected. With all different settings, the cross correlation coefficient between different threads stays below 0.05 in Unique Key and ranges from 0.11 to 0.17 in Shared Key. Both approaches achieve significant service delay improvements. Fig.5 plots the CCDF of service delays for downloading 3MB files with 1MB chunks ($k = 3$) with $n = 3 \sim 6$, assuming all $n$ tasks in a batch start at the same time. In this setting, both approaches reduce 99th percentile delays by roughly 50%, 65% and 80% by downloading 1, 2 and 3 extra coded chunks. Although Shared Key demonstrates up to 3 times higher cross correlation coefficient, there is no meaningful statistical distinction in service delay between the two approaches until beyond 99th percentile. All availability regions experience different degrees of degradation at high percentiles with Shared Key due to the higher correlation. Significant degradation emerges from around 99.9th percentile and beyond in all regions except for "Sao Paulo", in which degradation appears around 99th percentile.

(3) Task delays are always lower bounded by some constant $\Delta \geq 0$ that grows roughly linearly as chunk size increases. This constant part of delay cannot be reduced by using more threads: see the flat segment at the beginning of the CCDF curves in Fig.4 and Fig.5. Since this constant portion of task delays is unavoidable, it leads to the negative effect of using larger $n$ since there is a minimum cost of system resource of $n\Delta$ (time×thread) that grows linearly in $n$. This cost leads to a reduced capacity region for using more redundant tasks, as illustrated in the example of Fig.1. We observe that the two approaches deliver almost identical total delays (queueing + service) for all arrival rates, in spite of the degraded service delay with Shared Key at very high percentile. So we only plot the results with Shared Key in Fig.1.

(4) Both the mean and standard deviation of task delays grow roughly linearly as chunk size increases. Fig.6 plots the measured mean and standard deviation of task delays in both approaches at different chunk sizes. Also plotted in the figures are least squares fitted lines for the measurement results. Notice that the extrapolations at chunk size = 0 are all greater than zero. We believe this observation reflects the costs of non-I/O-related operations in the storage cloud that do not scale proportionally to object size: for example, the cost to locate the requested object. We also believe such costs contribute partially to the minimum task delay constant $\Delta$.

### C. Model of Task Delays

Based on the aforementioned observations, we decide to use the Shared Key approach in TOFEC since its outstanding storage efficiency overweights the minimum degradation in delay. For the analysis present in the next section, we model the task delays as independently distributed random variables whose mean and standard deviation grow linearly as chunk size $B$ increases. More specifically, we assume the task delay $D_t$ for chunk size $B$ following distribution in the form of

$$D_t(B) \sim \Delta(B) + exp(\mu(B)), \tag{1}$$

where $\Delta(B) = \overline{\Delta} + \widetilde{\Delta}B$ captures the lower bound of task delays as in observation (3), and $exp(\mu(B))$ represents a exponential random variable that models the tail of the CCDF. The mean and standard deviation of the exponential tail both equal to $\frac{1}{\mu(B)} = \overline{\Psi} + \widetilde{\Psi}B$. With this model, constants $\overline{\Delta}$ and $\overline{\Psi}$ together capture the non-zero extrapolations of the mean and standard deviation of task delays at chunk size 0, and similarly, constants $\widetilde{\Delta}$ and $\widetilde{\Psi}$ together capture the rate at which the mean and standard deviation grow as chunk size increases, as in observation (4).

## IV. Design of TOFEC

For the analysis in this section, we group requests into classes according to the tuple `(type, size)`. Here `type` can be read or write, and can potentially be other type of operations supported by the cloud storage. Each type of operation has its own set of delay parameters $\{\overline{\Delta}, \widetilde{\Delta}, \overline{\Psi}, \widetilde{\Psi}\}$. Subscripts will be used to indicate variables associated with each class. We first introduce approximations for the expected queueing and service delays, assuming the FEC code used to serve requests of each class is predetermined and fixed (Section IV-A). Then we formulate an optimization problem whose objective is to minimize the expected total delay over all such static strategies with fixed FEC codes. We show that solutions to the non-convex optimization problem exhibit a nice property (Section IV-B):

> The optimal values of $n_i$, $k_i$ and $r_i$ can all be expressed as functions solely determined by $Q$ – the expected length of the request queue:
>
> $$n_i = N_i(Q), \quad k_i = K_i(Q) \quad \text{and} \quad r_i = R_i(Q).$$
>
> $N_i$, $K_i$ and $R_i$ are all strictly decreasing functions of $Q$.

This finding is then used as the guideline in the design of our backlog-driven adaptive strategy TOFEC (Section IV-C).

### A. Approximated Analysis of Static Strategies

Denote $J_i$ as the file size of class $i$. Consider a request of class $i$ served with an $(n_i, k_i)$ MDS code, i.e., $B_i = J_i/k_i$. First suppose *all $n_i$ tasks start at the same time*, i.e., $T_1 = T_{n_i}$. In this case, given our model for task delays, it is trivial to show that the expected service delay equals to

$$\begin{aligned} D_{s,i} &= \Delta_i(J_i/k_i) + \frac{1}{\mu_i(J_i/k_i)} \sum_{j=0}^{k_i-1} \frac{1}{n_i - j} \\ &\approxeq \Delta_i(J_i/k_i) + \frac{1}{\mu_i(J_i/k_i)} \ln \frac{n_i}{n_i - k_i} \\ &= \overline{\Delta}_i + \frac{\widetilde{\Delta}_i J_i}{k_i} + \left( \overline{\Psi}_i + \frac{\widetilde{\Psi}_i J_i}{k_i} \right) \ln \frac{r_i}{r_i - 1}. \quad (2) \end{aligned}$$

Also define the system usage (or simply cost) of a request as the sum of the amount of time each of its tasks being served by a thread[7]. When all tasks start at the same time, its expected system usage is (see Section IV of [3] for detailed derivation)

$$\begin{aligned} U_i &= n_i \Delta_i(J_i/k_i) + \frac{k_i}{\mu_i(J_i/k_i)} \\ &= \overline{\Delta}_i k_i r_i + \widetilde{\Delta}_i J_i r_i + \overline{\Psi}_i k_i + \widetilde{\Psi}_i J_i. \quad (3) \end{aligned}$$

Suppose class $i$ contributes to $p_i$ fraction of the total arrivals, then the average cost per request is $\overline{U} = \sum_i p_i U_i$. With $L$ simultaneously active threads, requests depart the system at rate $L/\overline{U}$ (request/unit time). In light of this observation, we approximate the request queue with an $M/M/1$ queue with service rate $L/\overline{U}$. In other words, given the composition of requests $\{p_i\}$ and the choice of code(s) $\{n_i, k_i\}$, the system capacity (the maximum supportable throughput) is approximated with $L/\overline{U}$. So the queueing delay in the original system at total arrival rate $\lambda$ is approximated by

$$D_q = \frac{1}{L/\overline{U} - \lambda} - \frac{1}{L/\overline{U}} = \frac{\lambda \overline{U}^2}{L(L - \lambda\overline{U})}, \quad (4)$$

and the expected length of the request queue is approximately

$$Q = \lambda D_q = \frac{(\lambda\overline{U})^2}{L(L - \lambda\overline{U})} = \frac{\overline{\lambda}^2}{L(L - \overline{\lambda})}. \quad (5)$$

Here $\overline{\lambda} = \lambda\overline{U} = \lambda \sum_i p_i U_i = \lambda \sum_i p_i (\overline{\Delta}_i k_i r_i + \widetilde{\Delta}_i J_i r_i + \overline{\Psi}_i k_i + \widetilde{\Psi}_i J_i)$. Noticing that given $\{p_i\}$, $L/\overline{U}$ is maximized when $n_i = 1$, $k_i = 1$, $\forall i$, we call this maximum value the (approximated) **full capacity** for that $\{p_i\}$.

We acknowledge that the above approximation is quite coarse, especially because tasks of the same batch do not start at the same time in general. However, remember the main objective of this paper is to develop a practical solution that can achieve the optimal delay-throughput trade-off. According to the simulation results, this approximation is sufficiently good for the purpose of this paper.

[7]The time a task $j$ being served is $D_{t,j}$ if it completes successfully, $X_{(k)} - T_j$ if it starts but is terminated preemptively, and 0 if it is canceled while waiting in the task queue.

### B. Optimal Static Strategy

Given total arrival rate $\lambda$ and composition of requests $\{p_i\}$, we want to find the best choice of FEC code for each class such that the total delay is minimized. Relaxing the requirement for $n_i$ and $k_i$ being integers, this is formulated as the following minimization problem[8]:

$$\begin{aligned} \min_{\{k_i, r_i\}} \quad & D_q + \sum_i p_i D_{s,i} \quad (*) \\ \text{s.t.} \quad & k_i > 0, \quad r_i \geq 1, \quad \overline{\lambda} < L. \end{aligned}$$

Notice that this is a non-convex optimization problem because the feasible region is not a convex set, due to the $k_i r_i$ terms in $\overline{\lambda}$. In general, non-convex optimization problems are difficult to solve. Fortunately, we are able to prove the following theorem according to which this non-convex optimization problem can be solved numerically efficiently.

***Theorem 1:*** For any given $\lambda$ and $\{p_i\}$, the non-convex optimization problem $(*)$ has a unique optimal solution, which satisfies the following for all $i$:

$$\frac{k_i(\overline{\Psi}_i k_i + \widetilde{\Psi} J_i)}{\overline{\Delta}_i k_i + \widetilde{\Delta}_i J_i} = \frac{J_i r_i (r_i - 1)}{\overline{\Delta}_i r_i + \overline{\Psi}_i} \left( \widetilde{\Delta}_i + \widetilde{\Psi}_i \ln \frac{r_i}{r_i - 1} \right), \quad (6)$$

$$\left( \frac{L}{L - \overline{\lambda}} \right)^2 - 1 = \frac{2L(\overline{\Psi}_i k_i + \widetilde{\Psi}_i J_i)}{k_i r_i (r_i - 1)(\overline{\Delta}_i k_i + \widetilde{\Delta}_i J_i)}. \quad (7)$$

*Proof:* See Appendix. ■

Observing that Eq.6 contains only delay parameters and file size of class $i$, so it should be always satisfied no matter what arrival rate $\lambda$ and request composition $\{p_i\}$ are. Solving Eq.6 alone gives a set of pairs $(k_i, r_i)$ that are the optimal choice of code for class $i$ **for some** $\lambda$ and $\{p_i\}$. Then solving Eq.7 within this set we obtain the optimal $k_i$ and $r_i$ as a function of $\overline{\lambda}$ **for all** combinations of $\lambda$ and $\{p_i\}$ such that $\overline{\lambda} = \lambda \sum_i p_i U_i$. Observing from Eq.5 that $\overline{\lambda} = L \left( \sqrt{Q^2 + 4Q} - Q \right)/2$, and with some simple calculus, we conclude that

***Corollary 1:*** The optimal values of $n_i$, $k_i$ and $r_i$ can all be expressed as strictly decreasing functions of $Q$:

$$n_i = N_i(Q), \quad k_i = K_i(Q) \quad \text{and} \quad r_i = R_i(Q). \quad (8)$$

### C. Adaptive Strategy TOFEC

The finding of Corollary 1 conforms to our intuition:

- At light workload (small $\lambda$), there should be little backlog in the request queue (small $Q$) and the service delay dominates the total delay. In this case, the system is not operating in the capacity-limited regime. So it is beneficial to increase the level of chunking and redundancy to reduce delay.
- At heavy workload (larger $\lambda$), there will be a large backlog in the request queue (large $Q$) and the queueing delay dominates the total delay. In this case, the system operates in the capacity-limited regime. So it is better to

[8]Notice that all classes share the same queueing delay. Also, we require $k_i > 0$ instead of $k_i \geq 1$ for a technicality to simplify the proof of the uniqueness of the optimal solution. We require $r_i \geq 1$ since $n_i \geq k_i$. $\overline{\lambda} < L$ is imposed for queue stability.

reduce the level of chunking and redundancy to support higher throughput.

More importantly, it suggests that it is sufficient to choose the FEC code solely based on the length of the request queue. The basic idea of TOFEC is to choose $n_i = N_i(q)$ and $k_i = K_i(q)$ for a request of class $i$, where $q$ is the queue length upon the arrival of the request. When this is done to all requests arrive into the system, it can be expected the average code lengths (dimensions) and expected queue length $Q$ satisfy Eq.8, hence optimal delay is achieved. In TOFEC, this is implemented with a threshold based algorithm, which can be performed very efficiently. For each class $i$, we first compute the expected queue length if $n_i = 1, ..., n_i^{max}$ is the optimal code length by

$$Q^N_{i,n_i} = N_i^{-1}(n_i). \tag{9}$$

Here $n_i^{max}$ is the maximum number of tasks allowed for a class $i$ request. Since $N_i$ is a strictly decreasing function, its inverse $N_i^{-1}$ is a well-defined strictly decreasing function. As a result, we have $Q^N_{i,1} > Q^N_{i,2} > \cdots > Q^N_{i,n_i^{max}} > 0$. Remember our goal is to use code length $n$ if queue length $q$ is around $Q^N_{i,n}$, so we want a set of thresholds $\{H^N_{i,n}\}$ such that

$$\begin{aligned} H^N_{i,1} > Q^N_{i,1} > H^N_{i,2} > Q^N_{i,2} > \cdots \\ \cdots > H^N_{i,n_i^{max}} > Q^N_{i,n_i^{max}} > H^N_{i,n_i^{max}+1} = 0, \end{aligned}$$

and will use $n$ such that $q \in [H^N_{i,n+1}, H^N_{i,n})$. In our current implementation of TOFEC, we use $H^N_{i,n} = \left(Q^N_{i,n} + Q^N_{i,n-1}\right)/2$ for $n = 2, \cdots, n_i^{max}$ and $H^N_{i,1} = \infty$. A set of thresholds $\{H^K_{i,k_i^{max}}\}$ for adaptation of $k_i$ is found in a similar fashion. The adaptation algorithm of TOFEC is summarized in pseudo-codes as below:

***TOFEC (Throughput Optimal FEC Cloud)***

**Initialization:** $\overline{q} = 0$

`request` arrives

1: $q \leftarrow$ queue length upon arrival of `request`
2: $i \leftarrow$ class that `request` belongs to
3: $\overline{q} \leftarrow \alpha\overline{q} + (1-\alpha)q$
4: Find $k \le k_i^{max}$ such that $\overline{q} \in [H^N_{i,k+1}, H^N_{i,k})$
5: Find $n \le n_i^{max}$ such that $\overline{q} \in [H^N_{i,n+1}, H^N_{i,n})$
6: $n \leftarrow \min(r_i^{max}k, n)$
7: Serve `request` with an $(n,k)$ code when it becomes HoL.

Note that in Step 6 we reduce $n$ to $r_i^{max}k$ if the redundancy ratio of the code chosen in the previous steps is higher than $r_i^{max}$ – the maximum allowed redundancy ratio for class $i$. Also, instead of comparing $q$ directly with the thresholds, we compare an exponential moving average $\overline{q} = \alpha q + (1-\alpha)\overline{q}$, with a memory factor $0 \le \alpha < 1$, against the thresholds to determine $n$ and $k$. The moving average is used to mitigate the transient variation in queue length so that $n$ and $k$ will not change too frequently. It is obvious that we only need to set $\alpha = 0$ in order to use instantaneous queue length $q$ for the adaptation since in this case $\overline{q} = q$.

## V. Evaluation

We now demonstrate the benefits of TOFEC's adaptation mechanism. We evaluate TOFEC's adaptation strategy and show that is outperforms static strategies with both constant and changing workloads, as well as a simple greedy heuristic that will be introduced later.

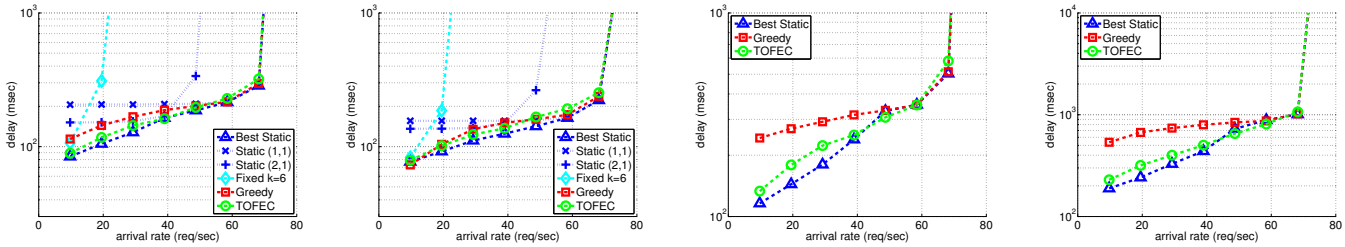
### A. Simulation Setup

We conducted trace-driven simulations for performance evaluation for both single-class and multi-class scenarios with both read and write requests of different file sizes. Due to lack of space, we only show results for the scenario with one class (`read,3MB`). But we must emphasize that it is representative enough so that the findings to be discussed in this section are valid for other settings (different file sizes, write requests, and multiple classes). We assume the system supports up to $L = 16$ simultaneously active threads. We set the maximum code dimension and redundancy ratio to be $k^{max} = 6$ and $r^{max} = 2$, because we observe negligible gain in service delay beyond this chunking and redundancy level from our measurements. We use traces collected in May and June in availability region "North California". In order to compute the threshold for TOFEC, we need estimations of the delay parameters $\{\overline{\Delta}, \widetilde{\Delta}, \overline{\Psi}, \widetilde{\Psi}\}$. For this, we first filter out the worst 10% task delays in the traces, then we compute the delay parameters from the least squares linear approximation for the mean and standard deviation of the remaining task delays. We use memory factor $\alpha = 0.99$ in TOFEC.

In addition to the static strategies, we develop a simple *Greedy* heuristic strategy for the purpose of comparison. Unlike the adaptive strategy in TOFEC, Greedy does not require prior-knowledge of the distribution of task delays, yet it achieves competitive mean delay performance. In Greedy, the code to be used to serve a request in class $i$ is determined by the number of idle threads upon its arrival: suppose there are $l$ idle threads, then $k_i = \begin{cases} 1, & \text{if } l = 0 \\ \min(k_i^{max}, l), & \text{otherwise} \end{cases}$, and similarly $n_i = \begin{cases} 1, & \text{if } l = 0 \\ \min(r_i^{max}k_i, l), & \text{otherwise} \end{cases}$. The idea of Greedy is to first maximize the level of chunking with the idle threads available, then increase the redundancy ratio as long as there are idle threads remain.

### B. Throughput-Delay Trade-Off

Fig.7 shows the mean, median, 90th percentile and 99th percentile delays of TOFEC and Greedy with Poisson arrivals at different arrival rate $\lambda$. We also run simulations with static strategies for all possible combinations of $(n,k)$ at every arrival rate. We brute-force find the best mean, median, 90th and 99th percentile delays achieved with static strategies and use them as the baseline. Also plot in Fig.7(a) and Fig.7(b) are the mean and median delay performance of the basic static strategy with no chunking and no replication, i.e., $(1,1)$ code; the simple replication static strategy with a $(2,1)$ code; and




(a) Mean Delay (b) Median Delay (c) 90th Percentile Delay (d) 99th Percentile Delay

Fig. 7. Delay performance in read only scenario

the backlog-based adaptive strategy from [3] with fixed code dimension $k = 6$ and $n \leq 12$.

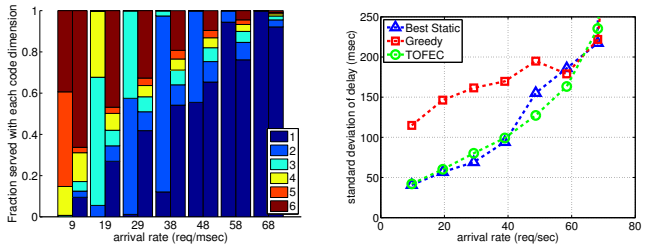
As we can see, both TOFEC and Greedy successfully support the full capacity region – the one supported by basic static – while achieving almost optimal mean and median delays throughout the full capacity region. At light workload, TOFEC delivers about $2.5\times$ improvement in mean delay when compared with the basic static strategy, and about $2\times$ when compared with simple replication (from 205ms and 151ms to 84ms). It also reduces the median delay by about $2\times$ from that of basic and simple replication (from 156ms and 138ms to 74ms). Meanwhile Greedy achieve about $2\times$ improvement in both mean (89ms) and median delays (79ms) over basic.

With heavier workload, both TOFEC and Greedy successfully adapt their codes to keep track with the best static strategies, in terms of mean and median delays. It is clear from the figures that both TOFEC and Greedy achieve our primary goal of retaining full system capacity, as supported by basic static strategy. On the contrary, although simple replication has slightly better mean and median delays than basic under light workload, it fails to support arrival rates beyond 70% of the capacity of basic. Meanwhile, the adaptive strategy from [3] with fixed code dimension $k = 6$ can only support less than 30% of the original capacity region, although it achieves the best delay at very light workload.

While the two adaptive strategies have similar performance in mean and median, TOFEC outperforms Greedy significantly at high percentiles. As Fig.7(c) and Fig.7(d) demonstrate, TOFEC matches with the best static strategies at 90th and 99th percentile delays throughout the whole capacity region. On the other hand, Greedy fails to keep track of the best static performance at lower arrival rates. At light workload, TOFEC's is over $2\times$ and $2.5\times$ better than Greedy at 90th and 99th percentiles. Less interesting is the case with heavy workload when the system is capacity-limited. Hence both strategies converge to the basic static strategy using mostly $(1, 1)$ code, which is optimal at this regime.

### C. Behavior of the Adaptation Mechanisms

When we look into the fraction of requests served by each choice of code, TOFEC and Greedy turn out to behave quite differently. In Fig.8(a) we plot the compositions of requests served by different code dimension $k$'s. At each arrival rate, the two bars represent TOFEC and Greedy. For each bar, the colors represent the fraction of requests served with code




(a) Composition of $k$. Left: TOFEC, Right: Greedy (b) Standard Deviation

Fig. 8. Comparison of TOFEC and Greedy

dimension 1 through 6, from bottom to top. TOFEC's choice of $k$ demonstrates a high concentration around the optimal value: at all arrival rate, over 80% requests are served by 2 neighboring values of $k$. Moreover, as arrival rate varies from low to high, TOFEC's choice of $k$ transitions quite smoothly as $(5, 6) \to (3, 4) \to (2, 3) \to (1, 2)$ and eventually converges to a single value 1 as workload approaches system capacity. On the contrary, Greedy tends to round-robin across all possible choices of $k$ and majority of requests are served by either $k = 1$ or 6. So Greedy is effectively alternating between the two extremes of no chunking and very high chunking, instead of staying around the optimal. Such "all or nothing" behavior results in $2\times$ to $3\times$ worse standard deviation as shown in Fig.8(b). So TOFEC provides much better QoS guarantee.

We further examine how well the two adaptive strategies adjust to changes in workload. In Fig.9 we plot the total delay experienced by requests arriving at different times within a 600-second period. The arrival rate is 10 request/second for the first and last 200 seconds, and 70 request/second for the middle 200 seconds. Both adaptive strategies turn out to be quite agile to changes in arrival rate and quickly converge to a good composition of codes that delivers optimal delays. On the contrary, the static strategy using $(3, 2)$ code builds up a huge backlog during middle 200-second period and takes over 100 seconds to clean it up.

## VI. Related Work

FEC in connection with multiple paths and/or multiple servers is a well investigated topic in the literature [7], [8], [9], [10]. However, there is very little attention devoted to the queueing delays. FEC in the context of network coding or coded scheduling has also been a popular topic from the perspectives of throughput (or network utility) maximization and throughput vs. service delay trade-offs [11], [12], [13],

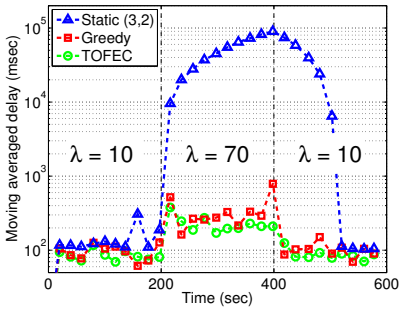



Fig. 9. Adaptation to changing workload

[14]. Although some incorporate queuing delay analysis, the treatment is largely for broadcast wireless channels with quite different system characteristics and constraints. FEC has also been extensively studied in the context of distributed storage from the points of high durability and availability while attaining high storage efficiency [15], [16], [17].

Authors of [4] conducted theoretical study of cloud storage systems using FEC in a similar fashion as we did in our work [3]. Given that exact mathematical analysis of the general case is very difficult, authors of [4] considered a very simple case with a fixed code of $k = 2$ tasks. Shah et al. [5] generalize the results from [4] to $k > 2$. Both works rely on the assumption of exponential task delays, which hardly captures the reality. Therefore, some of their theoretical results cannot be applied in practice. For example, under the assumption of exponential task delays, Shah et al. have proved that using larger $n$ will not reduce system capacity and will always improve delay, contradicting with simulation results using real-world measurements in [3] and this paper.

## VII. Conclusion

TOFEC's adaptation mechanism is the first technique for automatically adjusting the level of both chunking and redundancy for scalable key-value storage access using erasure codes and parallel connections. TOFEC monitors the local backlog and dynamically adjust both the length and dimension of the erasure code to be used. To evaluate TOFEC's adaptation mechanism, we run simulations using real-world traces obtained on Amazon S3. We found that TOFEC delivers the optimal delay-throughput tradeoff and dramatically outperforms non-adaptive strategies and simple adaptive heuristics.

## References

[1] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *USENIX ATC*, 2012.

[2] S. L. Garfinkel, "An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS," Harvard University, Tech. Rep., 2007.

[3] G. Liang and U. C. Kozat, "FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage with Coding," *IEEE/ACM Trans. Networking*, preprint, 13 Nov. 2013, doi: 10.1109/TNET.2013.2289382.

[4] L. Huang, S. Pawar, H. Zhang, and K. Ramchandran, "Codes Can Reduce Queueing Delay in Data Centers," in *IEEE ISIT*, 2012.

[5] N. B. Shah, K. Lee, and K. Ramchandran, "The MDS Queue: Analysing Latency Performance of Codes and Redundant Requests," *arXiv:1211.5405*, Apr. 2013.

[6] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren, "Stout: an Adaptive Interface to Scalable Cloud Storage," in *USENIX ATC*, 2010.

[7] V. Sharma, S. Kalyanaraman, K. Kar, K. K. Ramakrishnan, and V. Subramanian, "MPLOT: A Transport Protocol Exploiting Multipath Diversity Using Erasure Codes," in *IEEE INFOCOM*, 2008.

[8] E. Gabrielyan, "Fault-Tolerant Real-Time Streaming with FEC thanks to Capillary MultiPath Routing," *Computing Research Repository*, 2006.

[9] J. W. Byers, M. Luby, and M. Mitzenmacher, "Accessing Multiple Mirror Sites in Parallel: Using Tornado Codes to Speed Up Downloads," in *IEEE INFOCOM*, 1999.

[10] R. Saad, A. Serhrouchni, Y. Begliche, and K. Chen, "Evaluating Forward Error Correction performance in BitTorrent protocol," in *IEEE LCN*, 2010.

[11] A. Eryilmaz, A. Ozdaglar, M. Medard, and E. Ahmed, "On the Delay and Throughput Gains of Coding in Unreliable Networks," *IEEE Trans. Inf. Theor.*, 2008.

[12] W.-L. Yeow, A. T. Hoang, and C.-K. Tham, "Minimizing Delay for Multicast-Streaming in Wireless Networks with Network Coding," in *IEEE INFOCOM*, 2009.

[13] T. K. Dikaliotis, A. G. Dimakis, T. Ho, and M. Effros, "On the Delay of Network Coding over Line Networks," *Computing Research Repository*, 2009.

[14] U. C. Kozat, "On the Throughput Capacity of Opportunistic Multicasting with Erasure Codes," in *IEEE INFOCOM*, 2008.

[15] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Trans. Inf. Theor.*, 2010.

[16] R. Rodrigues and B. Liskov, "High Availability in DHTs: Erasure Coding vs. Replication," in *4th International Workshop, IPTPS*, 2005.

[17] J. Li, S. Yang, X. Wang, and B. Li, "Tree-Structured Data Regeneration in Distributed Storage Systems with Regenerating Codes," in *IEEE INFOCOM*, 2010.

## Appendix

*Proof:* The objective of $(*)$ is a lower-bounded continuously differentiable function within the feasible region. Its value goes to $\infty$ as $(k, r)$ approaches the boundary of the feasible region. As a result, there exist at least one global optimal solution. At the global optimal, derivatives of the objective over $k_i$ and $r_i$ both equal to 0. Equating the partial derivatives to 0 can be rewritten into Eq.6 and Eq.7. It is trivial to show that the left hand side of Eq.6 is a strictly increasing function of $k_i$ and the right hand side is a strictly increasing function of $r_i$ as long as $r_i \geq 1$. This implies that, $r_i$ is a strictly increasing function of $k_i$. The right hand side of Eq.7 becomes some function $\pi_i(k_i)$ of $k_i$ by substituting $r_i$ with the solution from Eq.6. It can be shown that $\pi_i$ is a strictly decreasing function. Notice that Eq.7 must be satisfied for all $i$ and the left hand side remains unchanged. Then

$$\pi_i(k_i) = \pi_j(k_j),\ \forall i, j = 1, \cdots, m,\ \forall i, j. \qquad (10)$$

Recall that $\pi_i$ and $\pi_j$ are strictly decreasing functions of $k_i$ and $k_j$, respectively. This means that there is a one-to-one mapping between any $k_i$ and $k_j$ at the optimal solutions, and $k_j$ is a strictly increasing function of $k_i$.

Notice that for any given $\lambda$ and $\{p_i\}$ the left hand side of Eq.7 is a strictly increasing function of $k_i$ if we replace all $k_j$'s and $r_j$'s with the solutions of Eq.6 and Eq.10. The right hand side of Eq.7 is $\pi_i(k_i)$, which is a strictly decreasing function of $k_i$. As a result, these two functions can be equal for at most one value of $k_i$, i.e., Eq.6 and Eq.7 have at most one solution. Since we have already proved the existence of a solution to these equations via the existence of global optimal, they have an unique solution. ■