Computer Networks 53 (2009) 110–125

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Algorithms and performance of load-balancing with multiple hash functions in massive content distribution

Ye Xia [*], Shigang Chen, Chunglae Cho, Vivekanand Korgaonkar

Computer and Information Science and Engineering Department, University of Florida, P.O. Box 116120, Gainesville, FL 32611-6120, United States

ARTICLE INFO

Article history:
Received 18 January 2008
Received in revised form 29 September 2008
Accepted 2 October 2008
Available online 17 October 2008

Responsible Editor: Y.C. Hu

Keywords:
Content distribution network
Load-balancing
Distributed-hash-table
Peer-to-peer network
File-sharing

ABSTRACT

We consider a two-tier content distribution system for distributing massive content, consisting of an infrastructure content distribution network (CDN) and a large number of ordinary clients. The nodes of the infrastructure network form a structured, distributed-hash-table-based (DHT) peer-to-peer (P2P) network. Each file is first placed in the CDN, and possibly, is replicated among the infrastructure nodes depending on its popularity. In such a system, it is particularly pressing to have proper load-balancing mechanisms to relieve server or network overload. The subject of the paper is on popularity-based file replication techniques within the CDN using multiple hash functions. Our strategy is to set aside a large number of hash functions. When the demand for a file exceeds the overall capacity of the current servers, a previously unused hash function is used to obtain a new node ID where the file will be replicated. The central problems are how to choose an unused hash function when replicating a file and how to choose a used hash function when requesting the file. Our solution to the file replication problem is to choose the unused hash function with the smallest index, and our solution to the file request problem is to choose a used hash function uniformly at random. Our main contribution is that we have developed a set of distributed, robust algorithms to implement the above solutions and we have evaluated their performance. In particular, we have analyzed a random binary search algorithm for file request and a random gap removal algorithm for failure recovery.

1. Introduction

One of the distinct trends related to the Internet is that it is being applied to the transfer of more and more massive content. This can be packaged DVD movies that Hollywood sells online, long and high quality streaming content, e.g., recorded TV programming, long-running video conferencing sessions, mountains of scientific data and all other automatically collected data such as consumer, market or economic data. There are two basic approaches to the distribution of such content. The first is by an overlay, infrastructure content distribution network (CDN), such as Akamai [1], Coral [10] or CoBlitz [25], where the servers in the infrastructure network interact with each other to replicate or cache the file. Requesting clients retrieve the file from a nearby server. The second approach is by a peer-to-peer (P2P) file-sharing network, such as BitTorrent [4] or Gnutella [12]. In this approach, there is no distinction between an infrastructure node or a regular client. Each peer can download the file or pieces of the file from a number of peers in parallel, and at the same time, serve the file or its pieces to some other peers. This is known as *collaborative download* or *swarming.*

We believe that both content distribution approaches will continue to coexist. In particular, the infrastructure-based approach is essential for commercially viable content distribution, for reasons such as service reliability and quality, accountability, or security. Hence, we consider a two-tier content distribution system consisting of an infrastructure CDN and a large number of ordinary clients.

[*] Corresponding author. Tel.: +1 352 392 2714.
E-mail addresses: yx1@cise.ufl.edu (Y. Xia), sgchen@cise.ufl.edu (S. Chen), ccho@cise.ufl.edu (C. Cho), vk2@cise.ufl.edu (V. Korgaonkar).

1389-1286/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2008.10.003

The nodes of the infrastructure network form peering relationship as in a structured, distributed-hash-table-based (DHT) P2P network, such as Chord [35], CAN [30], Pastry [31] or Tapestry [38]. Each file is first placed in the CDN, and possibly, is replicated among the infrastructure nodes depending on its popularity, i.e., the number of requests from the clients. A client sends its file request to the CDN, which routes the request to a node that contains the file. The client can then retrieve the file from the CDN node. Advanced P2P distribution techniques, such as swarming or parallel download, can be used among the infrastructure nodes when replicating a file in the CDN, or between the infrastructure nodes and the clients when downloading a file. However, the clients are not required to serve downloaded files to other clients or CDN nodes, even though it is possible to do so.

In a large CDN for massive content distribution, it is particularly pressing to have proper load-balancing mechanisms to relieve server or network overload. The subject of the paper is on popularity-based file replication techniques using multiple hash functions on a DHT-based CDN. DHT has already been introduced into the ad-hoc file-sharing systems such as BitTorrent. Its introduction into the CDN is similarly motivated by many desirable characteristics of DHT-based structured networks, such as allowing fast resource location, decentralized massive computation or data access, large-scale resource sharing, simplified routing, ease of management, and improved service quality and fault-tolerance due to path redundancy. At a deeper level, DHT gives every node and every piece of resource (e.g., file) a numeric ID; DHT-based networks can be viewed as distributed data structures for managing these IDs. The desirable characteristics of DHT come from clever data structures.

The envisioned framework of a DHT-based CDN and a popularity-based file replication strategy is in contrast to traditional, unstructured CDNs such as Akamai [1], or web caching systems [34], which replicate the files at essentially all edge servers where demand exists.[1] The main motivation behind our framework is that the traditional CDNs do not scale well or become inefficient for massive content distribution, as partially discussed in [25]. We will justify this in more details in Section 2.1.

In a DHT-based network, file placement is done by inserting the file into the distributed-hash-table. More specifically, a hash function is first applied to the file (e.g., the file name) and the returned hash value becomes the file ID. The file is then placed at a node that owns the range of hash values containing the file ID. Searching for a file (or locating a node) is to obtain the hash value of the file (or the node, respectively) and to route a query with the hash value as the destination address. Thus, the combination of hashing and structured routing eliminates the need of query flooding or establishing file directories. Compared with the original DHT-based networks for P2P file-sharing networks, in our design of the DHT-based CDN, the files themselves instead of the file pointers are placed in the network according to their IDs.

The idea of file replication with multiple hash functions is that, if k replicas of a file are needed, we will hash the original file with (at least) k hash functions, obtain k file IDs, and place the file in k nodes. One of the challenges is that it is not easy to decide the number of hash functions needed, since it is file dependent. The strategy assumed by this paper is that a *large* number of hash functions are set aside, enough for the most demanded file (e.g., the number being $m = 2^{32}$). With respect to a particular file, how many of these functions are actually used depends on the popularity of the file. We expect that most of them are not used for the majority of the files.

The focus of this paper is not in re-discovering the idea of using multiple hash functions for file replication, but in solving the unavoidable technical problems related to the use and management of the hash functions. Two central problems addressed are: How does a node choose one of the *unused* hash functions when replicating a file, or one of the *used* hash functions when requesting a file? Many solutions may be possible. But, they usually come with different performance-complexity tradeoffs, which are often difficult to understand. The contribution of this paper is that we propose simple and robust solutions, and more importantly, we thoroughly analyze the solutions and demonstrate they have very good performance.

Specifically, our solution to the former problem is to choose the first unused hash function for file replication. Assuming the IDs of the hash functions are $1, 2, \ldots, m$, this rule leads to the following invariance: When k hash functions are being used, they must have IDs $1, 2, \ldots, k$. Our solution to the latter problem is to choose a used hash function uniformly at random. We develop distributed algorithms that implement the above solutions and evaluate their performance. A key algorithm is a random binary search algorithm. Furthermore, for robustness in coping with node failures or the dynamics of node arrival and departure, we invent a random gap removal algorithm and evaluate its performance.

The paper is organized as follows. In Section 2, we review related works. In Section 3, we describe three algorithms governing the use and management of the hash functions. In Section 4, we evaluate the performance of the algorithms. In Section 5, we describe simulation results that compare our multiple hash function-based load-balancing approach to on-demand caching plus replication at neighbors. The conclusion is drawn in Section 6.

2. Background

2.1. Content distribution techniques

Traditional CDNs such as Akamai [1] and Coral [10], or web caching systems [34] mainly aim at distributing smaller web-related files and are not quite suitable for distributing massive files (see [25]). They typically replicate all files that are requested by clients, regardless of the request frequencies, at every edge server to improve the response time perceived by the clients. For massive content, this approach does not scale well due to the limitation in the memory or disk cache, or network bandwidth. At the minimum, it is wasteful.

[1] This is roughly true for web CDNs if we ignore the business dynamics, such as the service contract.

Instead, one should use popularity-based replication, taking advantage of the skew in the popularity of different files, as proposed for other systems [17,36]. This approach requires efficient file location service, whereas earlier CDNs only require the DNS-based server location service. In this paper, we assume a DHT-based CDN for file placement and location service, as apposed to an unstructured network that requires query flooding, a centralized directory or more static DNS-like lookup service. Many recent distributed file/storage systems, file-sharing systems and CDNs start to employ the DHT for file lookup, for instance, Coral [10], CFS [9], PAST [32] and OceanStore [21], because it allows fast, decentralized lookup. Other desirable characteristics of DHT-based networks include simplified routing, a small network diameter, path redundancy, fault-tolerance, scalability, and ease of management. A sample of such networks include Chord [35], Tapestry [38], Pastry [31], CAN [30], Koorde [16], ODRI [23], Ulysses [20], and FISSIONE [22].

Since our CDN is a P2P network, file replication among the CDN nodes can employ many techniques used in P2P collaborative file distribution. The advantages of collaborative distribution over a conventional single-server scheme, or even a tree-based multicast scheme, have been well established. For instance, it avoids server or network overload, achieves higher throughput or faster distribution speed, and is more resilient to link failure, frequent node departure, and traffic fluctuation. A sample of P2P collaborative distribution systems include BitTorrent [4], SplitStream [5], FastReplica [7], Bullet [19], Bullet' [18], Slurpie [33], ChunkCast [8], CoBlitz [25] and Julia [3]. An abstract problem in many of these works is how to distribute a file to *all* nodes as fast as possible. This is different from our problem, which concerns the distribution of a file to a subset of the CDN nodes based on its popularity.

2.2. Existing load-balancing techniques

Relevant file replication strategies that have been proposed previously can be summarized into three categories: (i) caching, (ii) replication at neighbors or nearby nodes, and (iii) replication with multiple hash functions. A file can be cached at nodes along the route of the publishing message when it is first published, or more typically, at nodes along the routes of query messages when it is requested. In approach (ii) above, when a node is overloaded with the requests to a file, it replicates the file at its neighbors, or at other nodes that are close in the ID space such as the successors or neighbor's neighbors. CAN and Chord mainly use strategy (ii), complemented by (i) and (iii).[2] Tapestry uses strategy (ii) and (iii). Following the suggestions in Chord, CFS [9] replicates a file at k successors of the original server and also caches the file on the search path. PAST [32], which is a storage network built on Pastry, replicates a file at k numerically closest nodes of the original server and caches the file on the insertion and the search paths. In the Plaxton network in [26], the replicas of a file are placed at directly connected neighbors of the original server and it is shown that the time to find the file is minimized. The replication strategies used in Coral [10] and Beehive [28] belong to the class of strategy (ii). Both systems replicate an object at its neighbors along the lookup tree.

Each of these strategies has its advantages and disadvantages, and in real systems, they can be used in combination to complement each other. Caching is often simple and can improve the response time of the queries if done properly. However, a naive caching algorithm cannot be a complete solution to the load-balancing problem, because even a good cache hit ratio, say 80%, still leaves 20% of the requests going to the original server for the file, which may overload the server many times beyond its capacity. Replication at neighbors does not have the cache-miss problem, if the file is replicated at all neighbors of the original server. However, in most proposed structured P2P networks, the load to each of the neighbors is not evenly distributed. In general, it is difficult to achieve truly balanced load with this approach because the assignment of requests to nodes depends on many factors and is not tightly controlled. Furthermore, even after the nodal hotspot is removed, the routing hotspot may still remain because all requests are directed to some neighborhood of the original server.

The main advantage of replication with hash functions is that, with uniform hash functions, copies of the file are uniformly distributed over the network, and with uniform use of the hash functions, file requests are also uniformly distributed over the set of replication servers for the file. The disadvantage is that the response time for queries is increased, as we will see later. But, response time is not a serious concern for large download, and can also be improved by parallel requests. Genuine uniform hashing is not able to preserve locality information, which is useful for assigning clients to nearby servers. However, some form of uniform hashing that maintains the locality information is possible, but is considered outside the scope of this paper.

In [17,6] and [27], file replication is performed through multiple hash functions, which are organized in a tree. This results in the replication servers being organized into a tree. In our case, the replication servers have no topological relationship. The fine dynamic replication (FDR) strategy introduced in [37] is also based on multiple hash functions. FDR is implemented on dedicated servers called request redirectors, which maintain information on server availability for each object and server load. Such information may not be accurate at all time and may not be consistent among the redirectors, which may cause unnecessary overload on some servers. On the other hand, there is no need of dedicated servers with our algorithm. The only information each client has is the set of hash functions. Server availability can be found by random binary search and server load is monitored by the server itself, which is always accurate. Overall, our algorithms are highly decentralized and keep minimum state information without loss of accuracy of needed information, and therefore, should be more robust against failures and other contingencies.

Another complementary load-balancing technique is to migrate the files in a heavily-loaded server to lightly loaded

[2] File replication in these and other structured networks is also (sometimes mainly) for the purpose of fault-tolerance.

servers [9,29,13]. There, the nodal overload is caused by having too many files mapped to the node rather than by having too many requests for one or a few files. Since those papers focus on a particular kind of nodal hotspot problem that is different from our file hotspot problem, their technique is orthogonal but can be complementary to our algorithms.

3. File replication, request and failure recovery algorithms

Our goal is to replicate a popular file into multiple copies and store them in different nodes, with the help of m uniform hash functions, denoted by $h_1, h_2, \ldots, h_m$, where m is a large enough number, say 2^{32}, so that no file will ever need more than m copies. It is not hard to have a family of such functions. One way is to use one hash function, h, but append a number i to the argument of the hash function, where $i = 1,2,\ldots,m$. For instance, if the argument is the file name, *foo*, then $h(foo1), h(foo2), \ldots, h(foom)$ gives m hash values for the file. The number i is called the "salt" value in [38]. For an in-depth discussion on creating proper hash functions, the readers are referred to [17].

Since m is a very large number, we do not want to replicate every file m times. Instead, we will take the popularity-based replication strategy [17,36]. The basic idea is that, every node keeps track the popularity of each file and replicates the file when the number of requests exceeds a threshold. As a result, the number of replicas produced depends on the popularity of the file. The focus of the paper is on hash function usage and management under this file replication strategy. One must consider two important questions. First, when requesting a file, how does the client quickly find a used hash function? Second, when the current servers cannot handle the requests for the file, how does the network replicates the file to other nodes, with the help of the unused hash functions?

With respect to a fixed file, let us call a CDN node that already contains a copy of the file a *filled node*. Otherwise, it is called an *empty node*. In the so-called *push* strategy, file replication is initiated by an overloaded filled node: It attempts to push a copy of the file to an empty node. Alternatively, upon seeing many requests, an empty node can locate a filled node and make a copy of the file. This is called the *pull* strategy. In practice, both strategies should be combined. We will mainly explain the algorithms in the push strategy since they are more complex and also provide the essential ingredients for the pull strategy.

3.1. Replication algorithm

This sub-section gives an answer to the second question raised above. The goal of our file replication algorithm is that, if k hash functions are used for replication, they must be $h_1, h_2, \ldots, h_k$. The rationale for this will become apparent when we discuss how to use the hash functions to access the file in Section 3.2. With this goal, in order to push a file to an empty node, the overloaded filled node must first find an unused hash function. Again, assume k hash functions are currently used, $h_1, \ldots, h_k$. The filled node must discover the number k and use the hash function h_{k+1}. It can do so by executing binary search for k between 1 and m, which takes $O(\log m)$ steps. More specifically, the node runs the `find_k(f, 1, m)` algorithm, to find the number k, where f is the file. Recall that the binary search algorithm maintains the current search interval $s, s+1, \ldots, t$, where, in the first search step, $s = 1$ and $t = m$. In each step, the algorithm tries to find out if h_i is used, where $i = \lfloor (s+t)/2 \rfloor$.[3] This is accomplished by routing the query with the $h_i(f)$ as the destination address in the infrastructure CDN. If the result of the query indicates that h_i is not used (i.e., file f is not present at the node that owns $h_i(f)$.), the original node calls `find_k(f, s, i)` and t is set to be i. On the other hand, if the result of the query indicates that h_i is used, the original node calls `find_k(f, i, t)`, and s is set to be i.

There are a number of well-known ways for a node to decide if it is overloaded. We assume a generic method: If the measured request rate is above a pre-defined threshold, then the node is overloaded.

3.2. File request algorithm: random binary search based hash function usage

We propose the following hash function usage scheme. First, the file replication algorithm ensures that the hash functions are used in increasing order of their indices. With this and assume that hash functions $h_1, \ldots, h_k$ are used for f, the goal of the file request algorithm is to choose one of the k used hash functions uniformly at random. Assuming each hash function maps the file f to a distinct node, then each filled node sees the same number of requests on average. To achieve this objective, the requesting node calls `search_f(f, m)`, shown in Algorithm 1, which is a random version of binary search. The function `uniform_random(l, u)` returns an integer between l and u, inclusive, uniformly at random. The function `query_nd(v)` returns the node that contains the hash value `v`.

Algorithm 1. search_f(f, u)

```
u ⇐ uniform_random(1, u);
nd ⇐ query_nd($h_u$(f));
if f exists at node nd then
  return nd
else if u == 1 then
  f cannot be found
else
  search_f(f, u)
end if
```

The idea of `search_f(f, u)` is that we first pick a random number between 1 and m, say i_1. If h_{i_1} is not used, in the next iteration, we pick a number between 1 and i_1 randomly, say i_2. If h_{i_2} is not used, in the next iteration, we pick a number between 1 and i_2 randomly. The algorithm goes on until a used hash function is returned or until it discovers that none of the hash functions is used.

[3] $\lfloor z \rfloor$ is the floor of the real number z, i.e., the largest integer not exceeding z.

3.3. Failure handling: the gap removal algorithm

The hash function usage scheme presented in Section 3.1 and 3.2 is adequate if no node ever leaves the P2P network unexpectedly. Otherwise, the files of the failed node may not be moved to appropriate nodes to ensure their continued availability. From the point of view of the hash functions, node failure may create gaps in the sequence of used hash functions.[4] Without proper repair, the gaps will accumulate over time and will likely cause the binary search algorithm to fail, undermining the effectiveness of the load-balancing scheme. For instance, imagine the case where the hash functions h_2 and h_4 are in use and h_1 and h_3 are not. Suppose, when applied to the file *f*, they each correspond to a different node. Then, there is a non-negligible probability that `search_f(f, m)` fails to return a replica server. In another example, suppose h_1 and h_4 are in use and h_2 and h_3 are not. Then, the node corresponding to h_1 will take a higher load than the one corresponding to h_4. This discussion suggests we should remove the gaps in the sequence of used hash functions.

We will consider the following simple gap removal algorithm. Let us focus on a particular file, say, *f*. For ease of discussion, we say the algorithm is run independently by each used hash function, whereas it is run by the node corresponding to the hash function. If, in practice, a filled node corresponds to multiple used hash functions, it should execute the algorithm separately on behalf of each used hash function. Every once in a while, a used hash function random selects another hash function with smaller index and checks if the latter is in use. If not, the latter function will be put to use and the former hash function is removed from usage. More specifically, suppose h_j is doing the checking. The filled node corresponding to $h_j(f)$, say node *A*, draws a number *I* randomly from $1, 2, \ldots, j-1$, then sends a query message for *f* to $h_I(f)$. If h_I is not in use, indicated by the fact that the node corresponding to $h_I(f)$, say node *B*, does not contain *f*, then a replica of *f* is created at node *B*. The replica at node *A* is removed, provided node *A* does not correspond to other used hash functions. With the filling of the gap corresponding to the missing function h_I and the removal of the hash function h_j from the used list, it appears that h_j is moved to h_I. This is illustrated in Fig. 1, where h_6 is removed and the gap at h_2 is filled.

It remains to specify the probability distribution that governs which hash function should be checked by the algorithm. We consider the following class of algorithm, called `compact(p)`, independently run by each used hash function. Let us focus on the *j*th hash function h_j. With probability *p*, it selects the hash function h_{j-1}, and with probability $1-p$, it selects a hash function uniformly at random from h_1 to h_{j-1}. If the selected hash function is used, then nothing is done. Otherwise, the selected hash function will be used and h_j will no longer be used. The special case of `compact(0)` is also known as the *uniform jump* algorithm. Both the uniform jump algorithm and `compact(0.5)` algorithm work well. But some tradeoffs are involved.

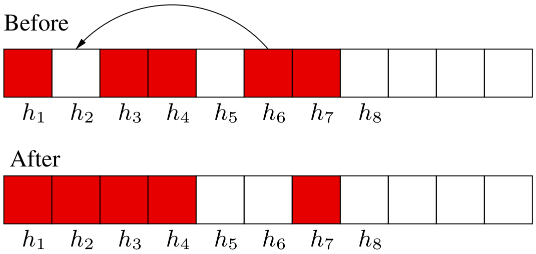
Before
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8
After
h_1 h_2 h_3 h_4 h_5 h_6 h_7 h_8

Fig. 1. Gap removal: remove h_6 and use h_2.

Primarily designed for failure recovery, the gap removal algorithm can also simplify the protocol executed when a node joins or leaves the CDN. No time-consuming file transfer is needed at the time of node arrival or departure. The gap removal algorithm will accomplish that at a later time.

3.4. Discussion

3.4.1. Centralized alternative

The essential assumption in our hash function management scheme is that the information about which hash functions are currently being used is not readily available. In our scheme, such information is accessed through distributed search. An obvious alternative is to keep the information at a centralized server. This scheme is not scalable if the centralized server is responsible for managing the hash functions of all files. For scalability, one can resort to a distributed database, either a DNS-like system or a DHT-based database. Besides the DHT-based system's lack of elegance – the DHT stores the hash functions used for another DHT that stores file content – both systems will have difficulty in keeping synchrony of the fast-changing hash function usage information across distributed servers. They also require non-trivial protocols or configuration procedures for handling failure or query overload. In contrast, our solution does not maintain a record (state) of the hash function usage information, but relies on fully distributed algorithms with minimum protocol support.

3.4.2. Parallel search algorithms

We will show later that the basic file request algorithm, Algorithm 1, takes about $\ln\frac{m}{k}$ search steps to find the file. For a large CDN, this may translate into delay of seconds. The delay can be reduced by a factor of *s* if a batch of *s* requests are sent out in parallel. In addition, the parallel algorithm is also useful for parallel download from the selected CDN servers to the requesting client.

4. Performance evaluation of the algorithms

4.1. Analysis on the random binary search algorithm

4.1.1. Hash function selection

Let $T(i)$ be the number of steps taken by Algorithm 1 to return a used hash function, assuming the first search step

[4] If a node fails unexpectedly, the query will land in a "neighboring" node determined by the DHT routing algorithm. The neighboring node may or may not have the corresponding file. In other words, the hash function gap may or may not exist.

takes place on the set $\{1, \ldots, i\}$, where $k \leqslant i \leqslant m$. We wish to find the statistics of $T(m)$. But, first, by conditioning on $T(m)$, it is easy to see that the returned function from the algorithm is chosen uniformly at random from h_1 to h_k.

The expected number of tries to find a used function and the variance are both $O(\log \frac{m}{k})$. The following theorems give the precise statements.

Theorem 4.1

$$\mathbf{E}T(m) = \begin{cases} 1 & \text{if } m = k, \\ 1 + \frac{1}{k} + \ldots + \frac{1}{m-1} & \text{if } m > k. \end{cases} \tag{1}$$

Proof. Conditional on the hash function returned from the first search step, we have the following iterative relation.

$$\mathbf{E}T(i) = \begin{cases} 1 & \text{if } i = k, \\ \frac{k}{i} + \frac{1}{i} \sum_{j=k+1}^{i} (1 + \mathbf{E}T(j)) & \text{if } k+1 \leqslant i \leqslant m. \end{cases} \tag{2}$$

The proof is by induction with the help of (2). □

By comparing the sum in Theorem 4.1 with integral, we get the following bounds for $\mathbf{E}T(m)$ for $1 < k < m$,

$$1 + \ln \frac{m}{k} \leqslant \mathbf{E}T(m) \leqslant 1 + \ln \frac{m-1}{k-1}. \tag{3}$$

Let Var(X) denote the variance of the random variable X. We can show

Theorem 4.2. *For $m > k$,*

$$\begin{aligned} \mathrm{Var}(T(m)) = \; & \frac{1}{k^2} + \frac{1}{(k+1)^2} + \ldots + \frac{1}{(m-1)^2} \\ & + \frac{1}{k} + \frac{1}{k+1} + \ldots + \frac{1}{m-1}. \end{aligned} \tag{4}$$

Proof. Again by conditioning on the hash function returned from the first search step, we have the following iterative relation, for $i = k+1, \ldots, m$.

$$\mathbf{E}T^2(i) = \frac{k}{i} + \frac{1}{i} \sum_{j=k+1}^{i} (1 + 2\mathbf{E}T(j) + \mathbf{E}T^2(j)). \tag{5}$$

With the help of (2) and (5), we can show inductively,

$$\begin{aligned} \mathbf{E}T^2(i) = & \left(1 + \frac{1}{k} + \frac{1}{k+1} + \ldots + \frac{1}{i-1}\right)^2 + \frac{1}{k^2} + \frac{1}{(k+1)^2} \\ & + \ldots + \frac{1}{(i-1)^2} + \frac{1}{k} + \frac{1}{k+1} + \ldots + \frac{1}{i-1} \quad \square \end{aligned} \tag{6}$$

For $1 < k < m$, reasonable bounds for Var($T(m)$) are

$$\begin{aligned} \ln \frac{m}{k} + \frac{1}{k} - \frac{1}{m} & \leqslant \mathrm{Var}(T(m)) \\ & \leqslant \ln \frac{m-1}{k-1} + \frac{1}{k-1} - \frac{1}{m-1}. \end{aligned} \tag{7}$$

For large m and $k \ll m$, $\mathrm{Var}(T(m)) \approx \ln \frac{m}{k}$.

In fact, it can be shown that $T(m) - 1$ can be approximated by a Poisson random variable with mean $\ln \frac{m}{k}$.

4.1.2. Access to all hash functions

In addition to load balance the file servers by choosing one of them uniformly for downloading, we also wish not to overload any node with excessive query traffic, even though the request message is much smaller than typical files. We have just established that each used hash function is selected with equal probability. However, the access pattern by the requests to the unused hash functions (i.e., the node corresponding to the hash function) in the random binary search algorithm is not uniform. Therefore, our next question is, by the end of the algorithm, how many times the hash function i has been accessed, where $k < i \leqslant m$.

To answer this question, we work with a continuous, scaled version of the algorithm for ease of analysis. In this version, consider the interval [0, 1] on which the interval [0, a] is marked, where $0 < a \leqslant 1$. The algorithm works similarly as Algorithm 1. Given the initial interval [0, 1], it performs random binary search until the region [0, a] is hit. More concretely, in the first search step, a number X_1 is chosen uniformly on [0,1]. If $X_1 > a$, in the second step, a number X_2 is chosen uniformly on $[0, X_1]$. Let the random variable T be the number of jump (search) steps taken before the algorithm returns some $y \in [0, a]$. Let X_i be the position of the ith jump in the algorithm, $i = 1, 2, \ldots$. Let us consider the stopped process, $X_1, X_2, \ldots, X_T$. For each $0 \leqslant y \leqslant 1$, let $N(y)$ be the number of X_i's less than or equal to y in the stopped process. That is

$$N(y) = |\{i : X_i \leqslant y, i = 1, 2, \ldots, T\}| = \sum_{i=1}^{T} 1_{(X_i \leqslant y)}.$$

where the indicator function $1_{(X_i \leqslant y)}$ is equal to 1 when $X_i \leqslant y$, and equal to 0 otherwise. Let $n(y) = \frac{d\mathbf{E}N(y)}{dy}$, and call it *hit density*. It is a kind of "density" in the sense that the expected number of hits by the requests on $[y, y + \Delta y]$ is $n(y)\Delta y$. It can be shown that

Theorem 4.3

$$n(y) = \begin{cases} \frac{1}{a} & \text{for } 0 \leqslant y \leqslant a \\ \frac{1}{y} & \text{for } a < y \leqslant 1 \end{cases}. \tag{8}$$

Proof. The proof is given in Appendix A. □

From the above theorem, we see that the un-marked region is hit less than the marked region per unit length. Translating this observation to the load-balancing application, we conclude that even though the unused hash functions are not accessed uniformly, each of them is accessed less than any of the used hash functions.

Due to the fact that the continuous version of the algorithm approximates the discrete version, Theorem 4.3 should also approximately apply to the discrete case. In Fig. 2, we plot the simulation results of hit counts to each hash function for the discrete algorithm, that is, the expected number of hits to each hash function by the time the algorithm finishes. In the same figure, we also show the function $1/n$, for $1 \leqslant n \leqslant m$ and the constant $1/k$. We see that Theorem 4.3 applies very well here.

In a separate note, Theorem 4.3 allows the pull-based replication strategy to be naturally integrated into our current framework. Since the query load is non-increasing as a function of the hash function index, the order of file replication in the pull strategy must correspond to the increasing order of the hash function index. This maintains the

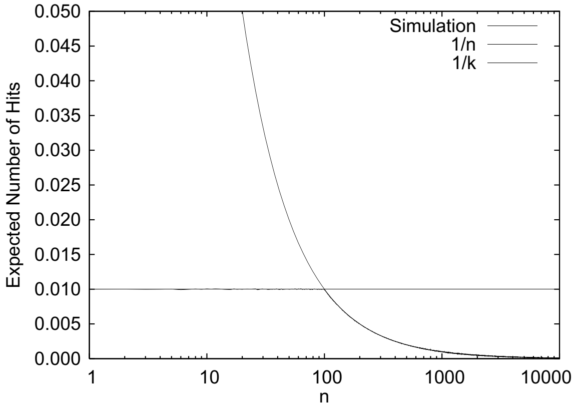

Fig. 2. Expected number of hits to each hash function, for m = 10,000 and k = 100.

key invariance of our framework that the hash functions are used in increasing order of their indices.

4.2. Analysis of the gap removal algorithm

Let us represent the status of the hash functions by a binary vector (or binary array) of length m, $x \in \{0,1\}^m$, with k 1's, where $1 \leqslant k \leqslant m$. Each 1 corresponds to a used hash function, and each 0 corresponds to an unused function. In accordance with the objective of the algorithm, we wish to move all 1's in x to the first k positions. That is, we'd like to compact x into the form 11...100...0. The question is how long this takes.

The discrete-time Markov chain embedded in the algorithm is equivalent to the following description. At each step, select one of the k 1's uniformly at random with probability $1/k$. Suppose the selected 1 is at position i, where $i \in \{1, 2, \ldots, m\}$, counted from the left to the right. With probability $g_i(j)$, we attempt to move the 1 to the left by j positions, where $1 \leqslant j < i$ and $g_i(j)$ satisfies $\sum_{j=1}^{i-1} g_i(j) = 1$ for each i. If the jth position to the left of position i is a 0, then the 1 is allowed to move. In other words, the 1 at position i becomes 0 and the 0 at position $i - j$ becomes 1, as if they exchange positions. Otherwise, the 1 is not moved.

Each state of the Markov chain is a m-digit binary vector, and a transition occurs every time a 1 attempts a move. Let us denote the finite-state Markov chain by $\{X_n\}_{n=0}^{\infty}$. The transition probability from state x to state y, denoted by $p(x, y)$, can be computed from the $g_i(j)$ functions above. Given the Markov chain starts at $X_0 = x$, the time it takes to finish compacting x is denoted by T_x. We write $v(x) = \mathbf{E}T_x$.

Starting with $X_0 = x$ and conditional on the first jump, we have

$$v(x) = \sum_y p(x, y)v(y) + 1. \tag{9}$$

Also, for the vector x = 11...100...0, we know that

$$v(x) = 0. \tag{10}$$

The solution to (9) and (10) exists and is unique (See Lemma 2 in chapter 4 of [11].). The problem can actually be solved efficiently due to its special structure. The difficulty lies in the potentially large dimension of the vector υ for large value of m.

We will consider some special vector types as the initial state under the uniform jump algorithm. The results shed light on the behavior of the algorithm for general cases.

4.2.1. Initial vector type: Isolated-1

An vector of the *Isolated-1* type starts (from the left) with consecutive 1's, followed by i consecutive 0's, followed by an isolated 1, then followed by zero or more 0's. An example is 1111000100 for i = 3. Let us index the vectors of the above form by, i, the number of 0's before the last 1, for i = 0, 1, ..., m–k. Clearly, $\upsilon(0)$ = 0. We can show that

Lemma 4.4

$$v(i) = \begin{cases} k^2 & i = 1 \\ k^2 + k\sum\limits_{j=2}^{i} \frac{1}{j} & 1 < i \leqslant m - k \end{cases}. \tag{11}$$

Proof. First, $\upsilon(1)$

$$v(1) = \frac{1}{k^2}(1 + v(0)) + \left(1 - \frac{1}{k^2}\right)(1 + v(1)) \tag{12}$$

The first term on the right hand side corresponds to the case where the isolated 1 is picked, with probability $\frac{1}{k}$, and moved to the only 0 to its left. The second term corresponds to all other cases. Rearranging (12), we get $v(1) = k^2$. Now suppose (11) is true for 1,2,...,$i - 1$, and we wish to show it is true for i. Conditional on the first jump, $\upsilon(i)$ satisfies

$$\begin{aligned} v(i) = &\frac{1}{k}\frac{1}{k + i - 1}(i + v(0) + v(1) + \cdots + v(i - 1)) \\ &+ \left(1 - \frac{i}{k(k + i - 1)}\right)(1 + v(i)) \end{aligned} \tag{13}$$

or

$$iv(i) = k(k + i - 1) + v(0) + v(1) + \cdots + v(i - 1) \tag{14}$$

Plugging into the above equation the expressions for $v(0), v(1), \ldots, v(i - 1)$, we get

$$\begin{aligned} iv(i) &= k(k + i - 1) + (i - 1)k^2 + \frac{k}{2}(i - 2) \\ &\quad + \frac{k}{3}(i - 3) + \cdots + \frac{k}{i - 1}(i - (i - 1)) \\ &= ik^2 + k(i - 1) + \frac{k}{2}i + \frac{k}{3}i + \cdots + \frac{k}{i - 1}i - k(i - 2) \\ &= ik^2 + k + \frac{k}{2}i + \frac{k}{3}i + \cdots + \frac{k}{i - 1}i \end{aligned} \tag{15}$$

Dividing both sides by i in the above, we have completed the proof. □

4.2.2. Initial vector type: Isolated-0

An vector of the *Isolated-0* type starts (from the left) with consecutive 1's, followed by exactly one isolated 0, followed by zero or more 1's, and then followed by zero or more 0's. In other words, it has the form 1...101...10...0. Let us reindex the vectors so that the ith vector has the isolated 0 at position $k - i + 1$, for i = 1,2,...,k. For instance, consider the case m = 5 and k = 3. The vectors 1, 2 and 3 are 11010, 10110 and 01110, respectively. Note that in the ith vector, the isolated 0 is followed by i consecutive 1's. For convenience, let us call the vector $1\ldots10\ldots0$ the 0th vector, and let $v(0) = 0$. We can show that

Lemma 4.5. *For* $i = 1, 2, \ldots, k$, $v(i) = k^2$.

Proof. We have already shown in the proof of Lemma 4.4 that $v(1) = k^2$. Suppose the lemma is true for $1, 2, \ldots, i-1$, where $1 \leqslant i < k$. We will show that it remains true for i. Conditional on the first jump, $v(i)$ satisfies

$$v(i)\frac{1}{k}\sum_{j=1}^{i}\frac{1}{k-i+j}(1+v(i-j)) + \left(1-\frac{1}{k}\sum_{j=1}^{i}\frac{1}{k-i+j}\right)(1+v(i)) \tag{16}$$

$$\sum_{j=1}^{i}\frac{1}{k-i+j}v(i) = k + \sum_{j=1}^{i}\frac{1}{k-i+j}v(i-j) \tag{17}$$

Using the induction hypothesis and the fact $v(0) = 0$, we get

$$\sum_{j=1}^{i}\frac{1}{k-i+j}v(i) = k + k^2\sum_{j=1}^{i-1}\frac{1}{k-i+j} = k^2\sum_{j=1}^{i}\frac{1}{k-i+j}$$

Hence, $v(i) = k^2$. □

We shall make some comments on the uniform jump algorithm. First, one should not be alarmed with the k^2 number of jump steps in Lemma 4.4 and 4.5, since the number of jumps per unit time scales linearly with k. Hence, the expected time it takes to complete the compacting process is linear in k. Second, uniform jump is suitable to quickly remove large gaps (long string of consecutive 0's). This is evident from the expression in (11), where the second term $\sum_{j=2}^{i}\frac{1}{j}$ is approximately ln (i). It is particularly suitable for the case where $k \ll m$ and the 1's in the vector concentrate at the right side of the vector, such as 000000000000111. Recall that the purpose of removing the gaps is for the binary search algorithm to quickly locate a used hash function (corresponding to a 1 in the vector). The aforementioned vectors are precisely those that most trouble the binary search algorithm. The uniform jump algorithm can quickly move the 1's toward the left side of the vector. Third, for vectors where the 1's concentrate at the left side, e.g., 101101111110000, the uniform jump algorithm is not very efficient in removing the last few 0's, particularly when k is reasonably large. This fact is evident from the k^2 term in Lemma 4.4 and 4.5. However, we are not very concerned with this because the binary search algorithm nonetheless will have a high chance of finding a 1 quickly for this type of vectors.

4.2.3. Simulation experiments for the gap removal algorithm

The above observations will be further supported by simulation experiments. In the simulation results, time is normalized in the following way. Each bit array (bit vector) entry with value 1, called a *marked entry*, makes a jump (compacting) attempt following a Poisson process, independently from other marked entries. The interval between any two consecutive attempts by the same marked entry, which is an exponential random variable, has mean 1 time unit. All durations are measured with respect to this time unit. Note that the average number of jump attempts by all marked entries in each time unit is equal to the number of the marked entries, i.e., the number of 1's in the array.

In the following, we will mainly consider the simulation results of the uniform jump algorithm, but will mention the performance tradeoffs that can be achieved by the `compact(0.5)` algorithm.

The initial array type to be considered is known as *Ones-at-End*, which has k consecutive 1's at the end of the array, following m–k 0's. An example is 00000111 for $m = 8$ and $k = 3$. In terms of the time required to finish compacting, one tends to believe that such array type represents the "worst case" for many compacting algorithms, including uniform jump. However, we have not proven this claim. Our extensive experiments have provided some evidence for the conjecture. For instance, Ones-at-End has slightly worse mean required time than another initial array type, *Random-Choice*, where the k marked entries are chosen uniformly at random from the set of indices $\{1,2,\ldots,m\}$ without replacement. This is shown in Fig. 3. Note the linear dependence of the mean completion time on k, the number of marked entries. If this is deemed as being too slow when k is large, we can address it in two different ways. First, it turns out that the compacting process becomes "nearly" finished much sooner than its completion. In other words, the array becomes useful, with respect to performing random binary search, much sooner than the completion time. Second, the `compact(0.5)` algorithm can be used, if desired, to make the dependence on k sub-linear, and hence, dramatically improves the mean completion time. The price to pay is increased delay before the probability of eventually hitting a marked entry reaches 1.

Recall that our objectives for compacting the binary array are to ensure, first, that the random binary search algorithm will eventually hit a marked entry and, second, that the load (or hitting probability) to each marked entry is balanced. Both objectives are fulfilled after the compacting process finishes. However, the probability of eventually hitting a marked entry can reach 1 long before the process finishes, as soon as the value in the first location of the

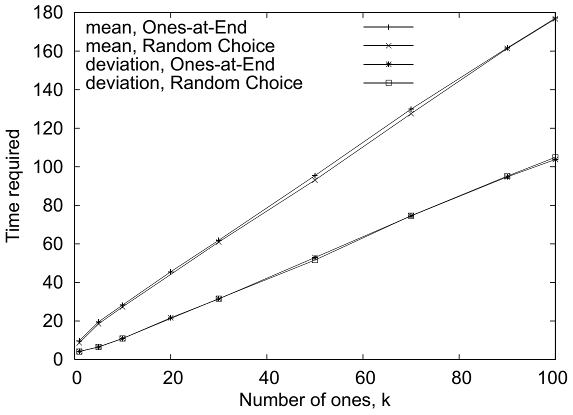

Fig. 3. Time required to compact the 1's. Comparison of ones-at-end and random-choice. $m = 10{,}000$.

array becomes 1. In Fig. 4, we show this probability as a function of time, while the compacting process is running, for three cases, $k = 10$, $k = 100$ and $k = 1000$. Each of these curves represents a typical sample path of the compacting process. Observe that the probability increases to 1 exponentially fast, well before the mean completion time of the compacting process, which is 28.27, 177.12 and 1665.62 for $k = 10$, $k = 100$ and $k = 1000$, respectively.

To examine how well our second objective of the compacting algorithm is fulfilled, in Fig. 5, we plot the load to each of the marked entries as the time progresses for the same instances as in Fig. 4. This is the hitting probability to each of the marked entries conditional on that at least one of them is hit. We see that, at time 0, the marked entries are hit uniformly. However, as seen from Fig. 4, the probability of an eventual hit to any marked entry is low. As the compacting algorithm operates, the uniform load pattern is first destroyed (however, the eventual hit probability increases), and then gradually restored. At time 15, the load is almost uniform except for the last few marked entries. Considering the fact that the mean completion time of compacting is 177.12 for this case, we see that the vast majority of the compacting time is dedicated to compacting the last few marked entries, while the other marked entries are already packed into appropriate places, as predicted by Lemma 4.4.

4.3. Overhead

Since our algorithms are used for massive content distribution where the file sizes are very large, communication overhead is in general negligible compared to the actual file transmission. For look up, our algorithm needs about ln (m/k) message transmissions in worst case, where m is the total number of hash functions and k is the number of hash functions actually used for the particular file being searched. Consider the worst case scenario, where $m = 2^{32}$ and $k = 1$. It takes $\ln(m/k) = 22$ lookup messages. Assume that the message size is 100 bytes, which may contain the source IP, port number, file name, and file ID. Assume also that the file size is 5 Gbytes. Then, the aggregate size of the transmitted control messages is $100 \times \ln(2^{32}) = 2.2$ Kbytes, which is nearly negligible compared to the file size. The parallel search algorithm increases the control message overhead by the number of parallel search messages, which is typically no more than 8.

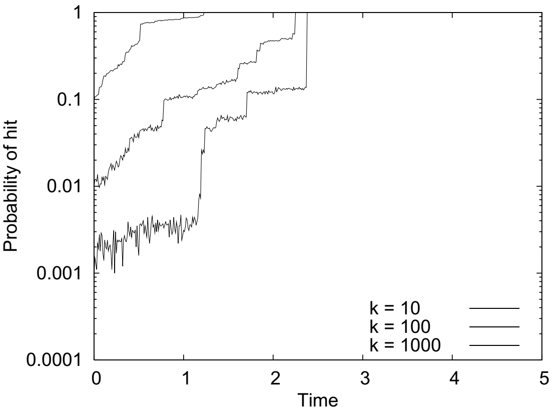

Fig. 4. Probability of eventually hitting a marked entry. The initial array is of the ones-at-end type. $m = 10{,}000$.

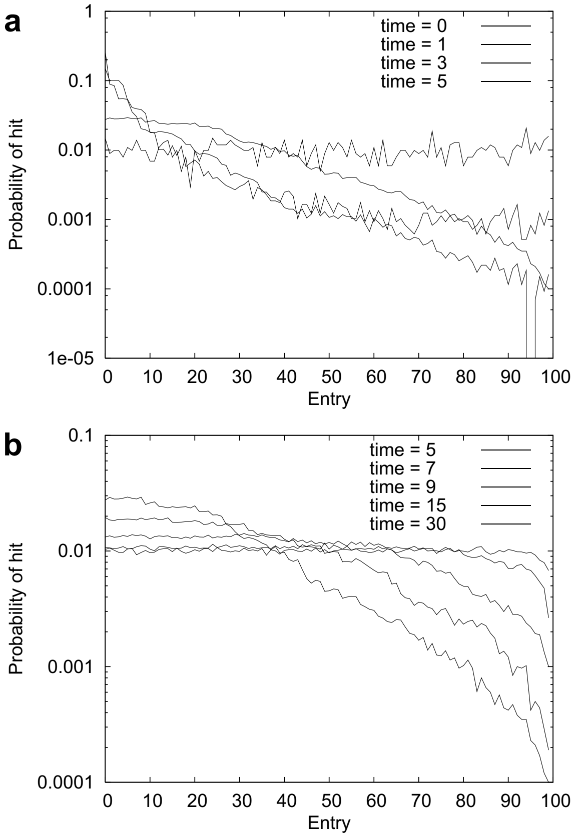

Fig. 5. Load to the marked entries over time. The initial array is of the ones-at-end type. $m = 10{,}000$, $k = 100$. (a) during time 0–5; (b) during time 5–30.

The communication overhead of the gap removal algorithm is also small. The reason is that the system still functions well with a small number of gaps: Even when some of servers containing a file are down accidentally, clients looking for the file can still find some other servers by continuing the random binary search. But, if the gaps keep accumulating without repair, the system performance will deteriorate. Hence, it is sufficient to run the gap removal algorithm in the background in a low-activity mode, for instance, once every 30 seconds or even every several minutes. This should be frequent enough for relatively stable CDNs, where node failures are infrequent.

The gap removal algorithm needs to be executed for each file replica, by the node containing the replica. The overall message overhead in the whole network is proportional to the total number of file replicas in the system. When the file popularity follows the Zipf distribution, most of the files don't need to be replicated, which means they don't need to execute the gap removal algorithm. Finally, a node with many replicas (for different files) can also adjust the running frequency based on the total number of replicas it contains.

5. Experiments

5.1. Comparison with other replication strategy

In this section, we present the simulation results with which we compare the replication strategy using multiple hash functions and random binary search (for brevity, called the MH strategy) with an on-demand caching strategy augmented by replication at neighbors. For both cases, the simulation is conducted on the Tapestry network. On Tapestry, an object (e.g., file) stored at a server is published along the publish path to a node known as the object root, which is uniquely determined by *surrogate routing*. The nodes along the publish path each have an object pointer to the server. A query is routed along the query path which is also determined by surrogate routing. Tapestry's routing guarantees that the query discovers a proper object pointer at a node on the publish path of the object, as long as the object exists in the network [38]. Tapestry's focus is on replication and caching of the object pointers, instead of the objects themselves. The main objective is to be able to locate each object pointer quickly. But, there is also a provision of on-demand caching of object content for load-balancing purpose.

In the simulation, the name space size is 2^{32}, the number of levels is 8, the size of each level is 16, and the number of nodes is 4096. This implies that each node ID is an eight-digit hexadecimal number, and that each node has up to eight primary neighbors, one at each level. To simulate the distance and delay between nodes, we assume every node has a physical position in a 1000×1000 square. The distance between a pair of nodes is the Euclidean distance, which determines the delay. Note that, for our purpose, it is sufficient to consider such an abstract distance model instead of a more realistic underlay network.

At the start of a simulation run, only one node has the file. We use a measurement interval of 10 time units (e.g., seconds, minutes) for each node to check the file request rate. We also define a request rate threshold that triggers a file replication event, which is 1. The meaning is that if the number of requests observed by a node is more than 10 per interval, then the node initiates a file replication. To prevent unnecessary replication caused by temporary fluctuation of the request rate, we use the exponentially weighted moving average of the request rate. We configure the simulator to generate requests randomly with a total rate of 40 requests per time unit. Ideally, 40 servers are needed with perfectly balanced load so that each server experiences a request rate exactly equal to the threshold.

When the MH strategy is integrated with the Tapestry network, the file is stored at its root node, which is determined by the primary file ID, $h_1(f)$, i.e., obtained by applying the first hash function to the unique file name. Therefore, we can replicate the file to up to m different nodes using m different hash functions. In our simulation, the MH strategy uses 128 hash functions for searching or replicating the file. If a server experiences a higher request rate than the threshold, it initiates file replication to another node, determined by the next available hash function.

Tapestry utilizes an on-demand caching strategy for relieving server overload, which is shown in Fig. 6. We name it the *caching along query path* (CQP) strategy. With the CQP strategy, if a node observes a higher request rate passing through it than the threshold, it requests a copy of the file to be cached locally, the source of which is some node along its own query path to the server. After the replication, the node becomes a new server that can intercept queries and serve the file from the local copy. CQP is a pull strategy because a non-server node initiates the replication to itself. The pulling-only CQP strategy has a problem that if a server experiences overload, it can only count on other nodes caching the file and intercepting sufficient queries. There is no guarantee that the overload can be resolved. Therefore, we extend CQP by combining it with a push replication strategy. Whenever a server detects an excessive request rate, it replicates the file to the neighbor that has sent the most requests during the current measurement interval. We call this integrated strategy CQP-push. Note that CQP-push combines caching on-demand and replication at neighbors.

We use the following metrics to compare the replication strategies.

- **Distribution of server load.** We measure the load of each server by counting the number of requests arriving at each server during the time intervals of interest. Each interval is between two consecutive file replication events. Note that the intervals do not have identical duration.
- **Worst case and deviation of server load.** We measure the average, maximum, minimum, and standard deviation of server load across the current servers at the current interval. We show the change of server load over time.
- **Final number of replicas.** We show the final number of replicas produced by each strategy. This number guarantees that none of the servers is overloaded. However, due to statistical fluctuation in the measurement-based algorithm, this number exceeds the minimum number required for each strategy. Since the file size is large, restricting the number of unnecessary replicas is a very important performance issue. It has implications in the

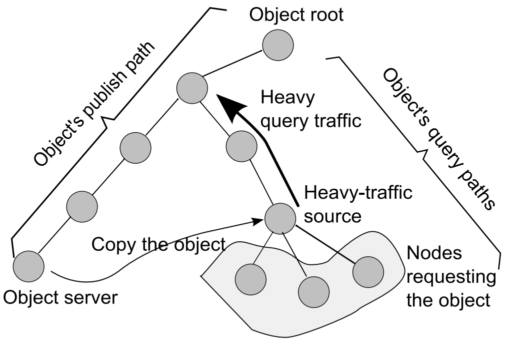

Fig. 6. On-demand caching in Tapestry. The object server replicates its object to the heavy-traffic source on query paths.

size of the distribution system required, including the network bandwidth requirement, the number of servers and their memory, disk and computational capacity.

5.1.1. Uniform requests throughout network

In this experiment, we compare the goodness of the replication strategies when the requests for the file are generated uniformly at random throughout the network. Fig. 7 shows how the server load decreases and how the number of replicas increases over time. For the server load, each narrow, vertical box represents one standard deviation above or below the average of the request rates seen by the servers on each measurement interval; the two ends of each vertical line represent the maximum and minimum request rate across servers. The maximum and the standard deviation of server load is much lower in MH than in CQP-push throughout time, indicating that the former achieves better server load-balancing. In the end, CQP-push needs much more replicas than MH, because its server load is not as uniform.

Fig. 8 shows the distribution of server load at the moment when the system has 50 replicas. CQP-push shows highly skewed distribution. This is because the servers encountered along the query paths but near the root node, which is the final destination of queries, may not be used frequently when their upstream nodes along the query paths also contain the file. On the other hand, MH shows quite even distribution, as theory predicts.

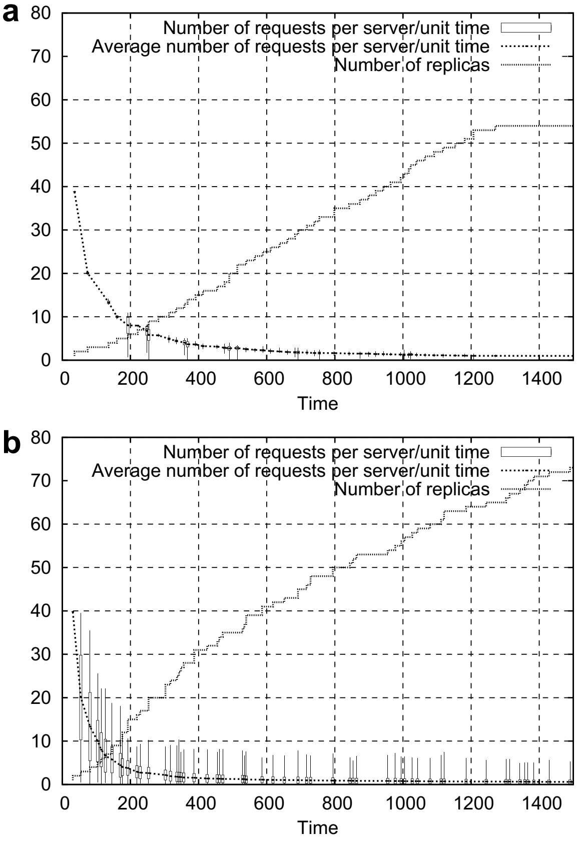

Fig. 7. The change of server load and the number of replicas over time. (a) MH and (b) CQP-push.

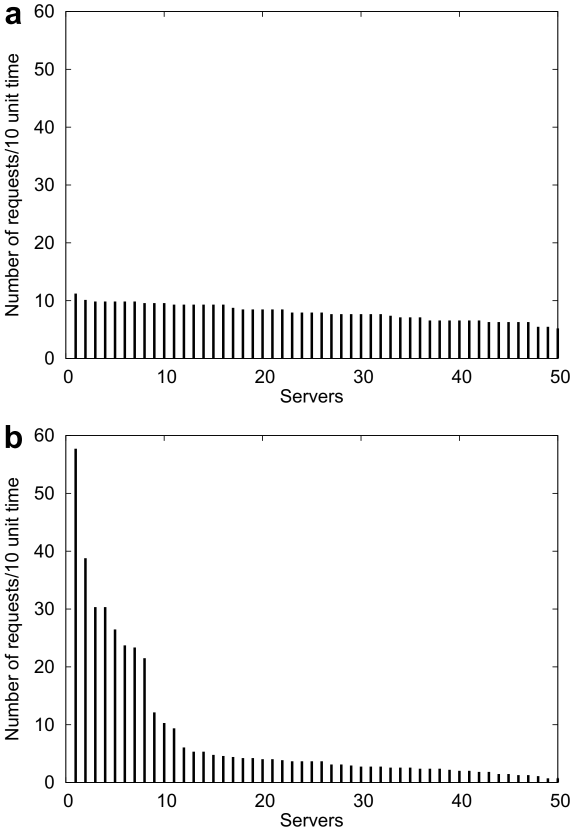

Fig. 8. Server load snapshot when the system has 50 file replicas. (a) MH and (b) CQP-push.

5.1.2. Localized requests from a region

We next compare the performance of the replication strategies when the request pattern is not uniform throughout the network. In reality, this can happen for many reasons that are difficult to foretell. To emulate the non-uniform request pattern, we restrict the requests to be generated uniformly from a region of the entire physical space. No requests are generated from outside the designated region. As shown in Fig. 9, with MH, the file replicas are distributed throughout the network at the end of simulation, whereas in CQP-push, they tend to group together in the request region. On one hand, having the servers closer to the request region brings the benefits of shorter round-trip time (RTT) and more localized data transfer. On the other hand, it causes higher network stress if the bandwidth in the region is not abundant. More importantly, we could expect that with CQP-push, if the request region moves from one to another, then most previous replicas may not be used, resulting in an unnecessarily large number of replicas.

Fig. 10 plots the change of server load and the number of replicas over time when the requests are generated from

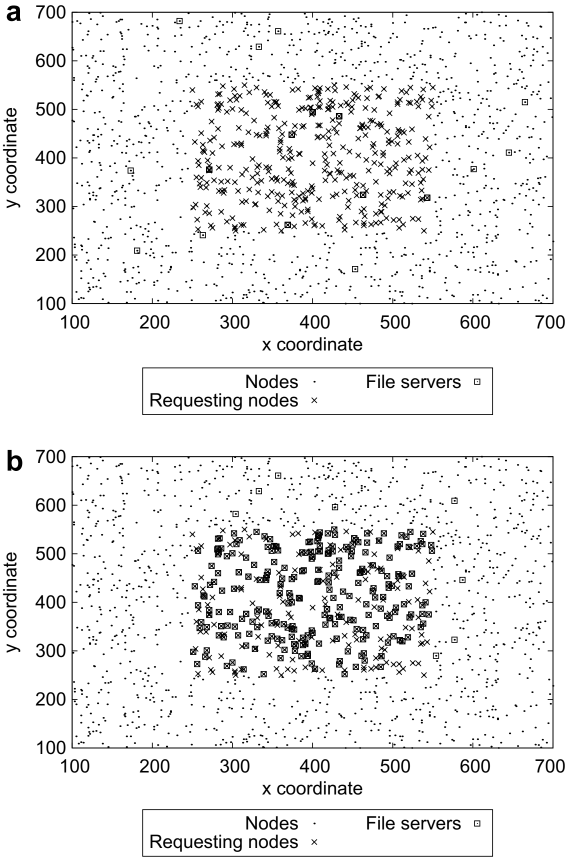

Fig. 9. Distribution of file servers at the end of simulation with localized requests. (a) MH and (b) CQP-push.

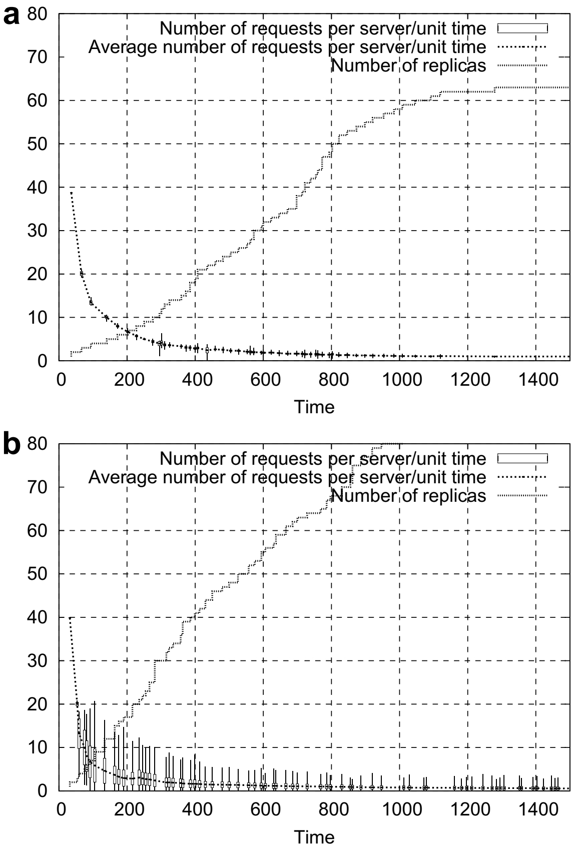

Fig. 10. The change of server load and the number of replicas over time with localized requests. (a) MH and (b) CQP-push.

a restricted region. Again, MH has much more balanced server load than CQP-push for all time.

Fig. 11 shows the distribution of server load after the replication process finishes. MH enjoys very well balanced server load whereas CQP-push has skewed distribution. In addition, CQP-push requires about four times as many replicas as MH. In a separate note, it also takes twelve times as long time as MH before the replication process ends.

5.2. Simulation with multiple files

The main algorithms of this paper have been developed by focusing on a single file. When there are multiple popular files, which is usually the case, the default strategy is to run the single file algorithms independently for each file. The question is whether this strategy leads to well balanced nodal load given that the load is now the aggregate of the per-file load over all the files contained by the node. The simulation results in this subsection will show that the strategy performs well. Next, we show that the performance can be further improved with an easy modification to the basic algorithms by allowing two random-choices for locating a node for replications and for queries.

In these experiments, we assume that the popularity of the files follows the Zipf distribution (which is widely assumed in CDNs). We use 1000 nodes and 10,000 files and generate up to 2.7 million queries. The targeted file of the query is determined by the Zipf distribution with a parameter 0.271, which is widely used in CDN simulations. At a node, the replication of a file is initiated whenever the number of requests to the file exceeds a threshold, which is set to 100.

We present the simulation results of the multiple file scenario with two different strategies, which mainly differ in the way of deciding where a file is replicated: the single-choice strategy and the multiple-choice strategy. The single-choice strategy is exactly the same as the one described in Section 3 using a single family of hash functions. The multiple-choice strategy uses two or more different hash function families for locating a node for replication and requests. Whenever a node needs to replicate a file, it locates two candidate destination nodes using two hash function families. Then, it compares the nodal loads on those nodes and chooses the node with lower load for replicating the file. Whenever a client requests a file, it uses both hash function families simultaneously, finds two nodes and compares the file-specific loads at the two nodes. The request will be served by the node with lower file-specific load.

The multiple-choice strategy can balance the load more effectively than the single-choice strategy. The reason can

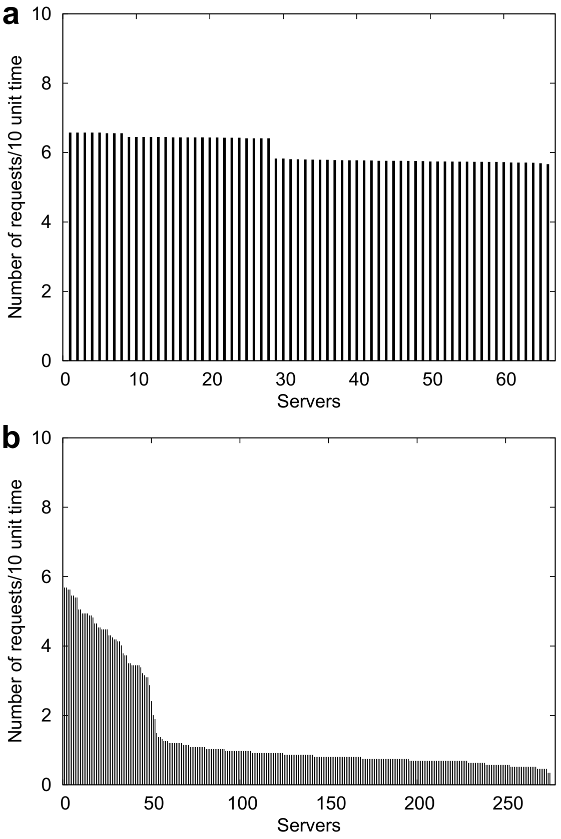

Fig. 11. Server load snapshot when the number of replicas becomes stable. (a) MH (66 replicas) and (b) CQP-push (277 replicas).

be explained by examining a related balls-in-bins (BNB) problem [15,14,2,24], which is an abstract model for load-balancing. The single-choice BNB problem is to place n balls into k bins by selecting one destination bin uniformly at random and independently across different balls. It is well known that, with $k = n$, the number of balls in the bin with the most balls, which corresponds to the maximum load, is $(1+o(1))\frac{\ln n}{\ln\ln n}$ with high probability. Note that the average number of balls per bin, which corresponds to the average load, is 1 in this case. In the multiple-choice BNB problem, for each new ball, d bins are independently and randomly selected and their contents are examined. The new ball is placed into the bin with the fewest balls. Rather surprisingly, for $d > 1$, the maximum becomes $(1+o(1))\frac{\ln\ln n}{\ln d}$ with high probability, an exponential reduction in the maximum load.

In Fig. 12a, we compare the nodal load distributions of the two strategies when the average nodal load is 1500 or 2700.[5] It shows that the single-choice strategy balances the load reasonably well, but the multiple-choice strategy does much better. For example, when the average load is 2700, 25.4% of the nodes each handles more than 3000 requests with the single-choice strategy, while the percentage drops down to 0.3% with the multiple-choice strategy. Fig. 12b shows the distributions of the number of files at each node under the two strategies. The multiple-choice strategy is also better in balancing the number of files over all nodes.

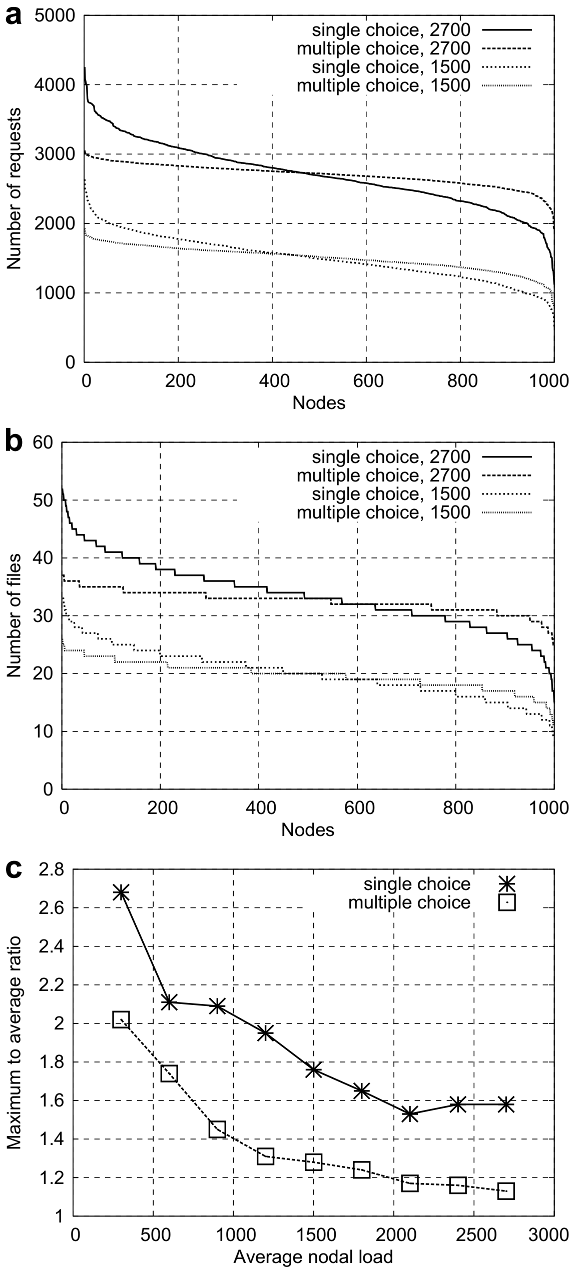

Fig. 12. Load and file distributions with multiple files: (a) Nodal load distribution; (b) number of files at each node and (c) maximum to average load ratio.

Fig. 12c shows the ratio of the maximum to average nodal load for both strategies over a wider range of average load. For both strategies, the ratio decreases fast initially as the average nodal load increases, but the decrease slows

[5] The nodal load is measured in terms of the number of file requests served. This measure makes sense if the file sizes are nearly identical. Here, we make this assumption for ease of presentation. The real system can in fact enforce this assumption by splitting a large file into multiple smaller files.

down eventually. The ratio in the multiple-choice strategy is consistently lower than that in the single-choice strategy and can get very close to 1.

Note that the experimental parameters we chose are quite pessimistic instead of optimistic. In other words, our experiments are about the bad cases. For instance, if the number of nodes is smaller while keeping other parameters unchanged, the load distribution over the nodes will be more balanced. This is the reason why the load balance improves as the average nodal load increases in Fig. 12c. One way to think about this is again to consider sending n balls (corresponding to file replicas) to k bins (corresponding to the nodes) uniformly at random. The average number of balls per bin is n/k and the maximum number of balls in any bin is roughly $n/k + \ln(k)$ with high probability, in the single-choice scheme. As k decreases, n/k dominates more and more the log term, which means the fluctuation of load becomes more and more negligible. In the multiple-choice scheme, the maximum number of balls in any bin is roughly $n/k + \ln(\ln n)$ with high probability, which is why it leads to even more balanced load. Hence, as k decreases, the load balance gets better.

The most imbalanced situation is when n = k, where each bin eventually has only 1 ball on average, but the fluctuation is $\ln n$ or $\ln(\ln n)$ in the single or multiple-choice schemes, respectively. In our experiments, when the total number of requests is 2.7 million over the interval of time (corresponding to the average nodal load 2700), we see from Fig. 12b that each node has about 35 files on average, including the original 10 files per node before replication. Hence, we have $k = 1000$, $n = (35 - 10) \times 1000 = 25,000$, and $n/k = 25$. When the total number of requests is 1.5 million (corresponding to the average node load 1500 in Fig. 12b), each node has 20 files on average in the end, which means it has 10 replicas. In this case, $k = 1000$, $n = 10{,}000$, and $n/k = 10$. Since the values of n/k are relatively small, the experimental results are close to the worst cases.

In the above simulation, we use per-file load for determining replication and assigning a request to a node. Nevertheless, the resulting nodal load is very well balanced. To further prevent unexpected nodal overload in rare contingency situations, our file load based replication strategy may be combined with a node load based strategy. In the latter strategy, if the aggregate requests for all its files exceed a threshold, the server replicates some of its files elsewhere, for instance, its most loaded files. We may also introduce admission control by which the overloaded server can reject additional file requests. Admission control is also a complementary solution to some other potential problems that have not been emphasized so far, such as many-to-one mapping from the hash functions to the servers and heterogeneous server capacities. It has been implemented in most P2P file-sharing applications. Finally, we may also adapt some other load-balancing techniques for resolving nodal hotspot, as introduced in [9,29,13].

6. Conclusion

This paper deals with algorithmic issues in file replication with multiple hash functions on DHT-based content distribution networks. The central issue here is, out of a potentially large number of hash functions, which one to use for downloading and which one to use for replicating a file so that the server load, and to some extent, the network load are balanced. Our main contributions are as follows. First, we have devised a complete set of algorithms for hash function usage and management. These include the random binary search algorithm for file request, the file replication algorithm and the hash function compacting algorithm for failure recovery. Second, we have thoroughly explored the performance of these algorithms by analysis and simulation. Third, we compare the proposed file replication scheme based on multiple hash functions with the combined scheme of on-demand caching and replication at neighbors.

Our algorithms for hash function usage and management are efficient, simple, and are compatible with the characteristics of the CDN we envision. These include the large network size, the massive content carried by the network, high infrastructure node dynamic, and a fast-changing file request pattern. In particular, the latter two characteristics make it difficult to run complicated protocols or to maintain consistency of state information kept at different nodes. Our solution to the file replication problem relies on fully distributed algorithms with minimum protocol support and without keeping any state information.

Appendix I. Appendix proof of Theorem 4.3

From $N(y) = \sum_{i=1}^{T} 1_{(X_i \leqslant y)}$, we have

$$\mathbf{E}N(y) = \sum_{i=1}^{T} P\{X_i \leqslant y\}. \tag{18}$$

When $0 \leqslant y \leqslant a$, $\mathbf{E}N(y) = P\{X_T \leqslant y\} = \frac{y}{a}$, and hence, $n(y) = \frac{1}{a}$. We will next focus on the case $a < y \leqslant 1$. In this case, $N(y) = \sum_{i=1}^{T-1} 1_{(X_i \leqslant y)}$. Conditional on the number of jumps by the algorithm, the hit density can be written as,

$$n(y) = \sum_{j=2}^{\infty} P\{T = j\} \sum_{i=1}^{j-1} p(X_i = y | T = j), \tag{19}$$

where $p(X_i = y|T = j)$ denotes the conditional density of X_i given $T = j$. To compute this conditional density, we start with the joint density. For $a < x_{j-1} \leqslant x_{j-2} \leqslant \ldots \leqslant x_1 \leqslant 1$,

$$\begin{aligned} &p(T = j, X_1 = x_1, \ldots, X_{j-1} = x_{j-1}) \\ &\quad = p(T = j | X_{j-1} = x_{j-1}) p(X_{j-1} = x_{j-1} | X_{j-2} = x_{j-2}) \ldots p(X_2 \\ &\quad = x_2 | X_1 = x_1) p(X_1 = x_1) = \frac{a}{x_{j-1}} \frac{1}{x_{j-2}} \cdots \frac{1}{x_1}. \end{aligned} \tag{20}$$

Hence,

$$\begin{aligned} &p(X_1 = x_1, \ldots, X_{j-1} = x_{j-1} | T = j) \\ &\quad = \frac{a}{x_{j-1}} \frac{1}{x_{j-2}} \cdots \frac{1}{x_1} / P\{T = j\}. \end{aligned} \tag{21}$$

Lemma 4.1. For $1 \leqslant i \leqslant j - 1$, *the marginal density*

$$p(X_i = x_i|T = j) = \frac{a}{P\{T = j\}}\frac{1}{x_i}\frac{1}{(j-1-i)!}\left(\ln\frac{x_i}{a}\right)^{j-1-i} \times \frac{1}{(i-1)!}\left(\ln\frac{1}{x_i}\right)^{i-1}. \tag{22}$$

Proof

$$\begin{aligned} p(X_i = x_i|T = j) &= \frac{a}{P\{T = j\}}\int_{a<x_{j-1}\leqslant\ldots\leqslant x_{i+1}\leqslant x_i\leqslant x_{i-1}\leqslant\ldots\leqslant x_1\leqslant 1}\frac{1}{x_{j-1}}\cdots\frac{1}{x_{i+1}} \\ &\quad\times\frac{1}{x_i}\frac{1}{x_{i-1}}\cdots\frac{1}{x_1}dx_{j-1}\ldots dx_{i+1}dx_{i-1}\ldots dx_1 \\ &= \frac{a}{P\{T = j\}}\frac{1}{x_i}\int_{x_i}^{1}\frac{dx_1}{x_1}\int_{x_i}^{x_1}\frac{dx_2}{x_2}\cdots\int_{x_i}^{x_{i-2}}\frac{dx_{i-1}}{x_{i-1}} \\ &\quad\times\int_a^{x_i}\frac{dx_{i+1}}{x_{i+1}}\int_a^{x_{i+1}}\frac{dx_{i+2}}{x_{i+2}}\cdots\int_a^{x_{j-2}}\frac{dx_{j-1}}{x_{j-1}}. \end{aligned} \tag{23}$$

By simple induction, it is easy to show

$$\int_a^{x_i}\frac{dx_{i+1}}{x_{i+1}}\int_a^{x_{i+1}}\frac{dx_{i+2}}{x_{i+2}}\cdots\int_a^{x_{j-2}}\frac{dx_{j-1}}{x_{j-1}} = \frac{1}{(j-1-i)!}\left(\ln\frac{x_i}{a}\right)^{j-1-i}, \tag{24}$$

and

$$\int_{x_i}^{1}\frac{dx_1}{x_1}\int_{x_i}^{x_1}\frac{dx_2}{x_2}\cdots\int_{x_i}^{x_{i-2}}\frac{dx_{i-1}}{x_{i-1}} = \frac{1}{(i-1)!}\left(\ln\frac{1}{x_i}\right)^{i-1}. \quad \square \tag{25}$$

For $a < y \leqslant 1$, combining the result of the Lemma 4.1 with (19), we have

$$\begin{aligned} n(y) &= \sum_{j=2}^{\infty}\frac{a}{y}\sum_{i=1}^{j-1}\frac{1}{(j-1-i)!}\left(\ln\frac{y}{a}\right)^{j-1-i}\frac{1}{(i-1)!}\left(\ln\frac{1}{y}\right)^{i-1} \\ &= \frac{a}{y}\sum_{j=2}^{\infty}\frac{1}{(j-2)!}\left(\ln\frac{y}{a}+\ln\frac{1}{y}\right)^{j-2} = \frac{a}{y}\exp\left(\ln\frac{y}{a}+\ln\frac{1}{y}\right) \\ &= \frac{a}{y}\frac{y}{a}\frac{1}{y} = \frac{1}{y} \end{aligned}$$

References

[1] Akamai Website <http://www.akamai.com>.

[2] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, Eli Upfal, Balanced allocations, SIAM Journal on Computing 29 (1) (2000) 180–200.

[3] D. Bickson, D. Malkhi, D. Rabinowitz, Efficient large scale content distribution, in: Proceedings of the Sixth Workshop on Distributed Data and Structures (WDAS'2004), Lausanne, Switzerland, July 2004.

[4] BitTorrent Website <http://www.bittorrent.com/>.

[5] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, A. Singh, Splitstream: High-bandwidth multicast in cooperative environments, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), October 2003.

[6] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael Schwartz, Kurt Worrell, A hierarchical internet object cache, in: Proceedings of USENIX Annual Technical Conference (USENIX'96), San Diego, CA, January 1996.

[7] L. Cherkasova, J. Lee, Fastreplica: efficient large file distribution within content delivery networks, in: Proceedings of the Fourth USITS, Seattle, WA, March 2003.

[8] ByungGon Chun, Peter Wu, Hakim Weatherspoon, John Kubiatowicz, Chunkcast: An anycast service for large content distribution, in: Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS), February 2006.

[9] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, Wide-area cooperative storage with CFS, in: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01), Banff, Alberta, Canada, October 2001, pp. 202–215.

[10] M.J. Freedman, E. Freudenthal, D. Mazières, Democratizing content publication with coral, in: Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 04), 2004.

[11] Robert G. Gallager, Discrete Stochastic Processes, Kluwer Academic Publishers, 1996.

[12] Gnutella Forums Website <http://www.gnutellaforums.com>.

[13] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, Ion Stoica, Load balancing in dynamic structured P2P systems, in: Proceedings of IEEE Infocom 2004, Hong Kong, March 2004.

[14] Gaston H. Gonnet, Expected length of the longest probe sequence in hash code searching, Journal of the ACM 28 (2) (1981) 289–304.

[15] N.L. Johnson, S. Kotz, Urn Models and Their Application, John Wiley and Sons, 1977.

[16] F. Kaashoek, D.R. Karger, Koorde: a simple degree-optimal hash table, in: Proceedings of the Second International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, CA, February 2003.

[17] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy, Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web, in: Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC'97), El Paso, TX, May 1997.

[18] D. Kostić, R. Braud, C. Killian, E. Vandekieft, J.W. Anderson, A.C. Snoeren, A. Vahdat, Maintaining high bandwidth under dynamic network conditions, in: Proceedings of USENIX Annual Technical Conference, 2005.

[19] D. Kostić, A. Rodriguez, J. Albrecht, A. Vahdat, Bullet: high bandwidth data dissemination using an overlay mesh, in: Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP'03), October 2003.

[20] A. Kumar, S. Merugu, J. Xu, X. Yu, Ulysses: a robust, low-diameter, low-latency peer-to-peer network, in: Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP03), Atlanta, Georgia, USA, November 2003.

[21] J. Kuviatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, Ramakrishna Gummadi, Sean Rhea, H. Weatherspoon, W. Weimer, B. Zhao, Oceanstore: an architecture for global-scale persistent storage, in: Proceedings of the Nineth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.

[22] Dongsheng Li, Xicheng Lu, Jie Wu, Fissione: a scalable constant degree and low congestion dht scheme based on kautz graphs, in: Proceedings of IEEE Infocom, Miami, FL, March 2005.

[23] D. Loguinov, A. Kumar, V. Rai, S. Ganesh, Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience, in: Proceedings of ACM SIGCOMM, Karlsruhe, Germany, August 2003.

[24] M. Mitzenmacher, The Power of Two Choices in Randomized Load Balancing. PhD Thesis, University of California, Berkeley, 1996.

[25] KyoungSoo Park, Vivek S. Pai, Scale and performance in the coblitz large-file distribution service, in: Proceedings of the Third USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 06), San Jose, CA, May 2006.

[26] C. Plaxton, R. Rajaraman, A. Richa, Accessing nearby copies of replicated objects in a distributed environment, in: Proceedings of the Nineth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97), Newport, Rhod Island, June 1997, pp. 311–320.

[27] C. Greg Plaxton, Rajmohan Rajaraman, Fast fault-tolerant concurrent access to shared objects, in: Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC'96), Philadelphia, PA, May 1996.

[28] Venugopalan Ramasubramanian, Emin Gun Sirer, Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays, in: Proceedings of the First Conference on Symposium on Networked Systems Design and Implementation, San Francisco, March 2004.

[29] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, Ion Stoica, Load balancing in structured P2P systems, in: Second International Workshop on Peer-to-Peer Systems (IPTPS'03), Berkeley, CA, February 2003, pp. 311–320.

[30] Sylvia Ratnasamy, Paul Francis, Mark Hanley, Richard Karp, Scott Shenker, A scalable content-addressable network, in: Proceedings of ACM SIGCOMM'2001, San Diego, CA, August 2001, pp. 161–172.

[31] Antony Rowstron, Peter Druschel, Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems, in: Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01), Heidelberg, Germany, November 2001.

[32] Antony Rowstron, Peter Druschel, Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility, in: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01), Banff, Alberta, Canada, October 2001, pp. 188–201.

[33] R. Sherwood, R. Braud, B. Bhattacharjee, Slurpie: a cooperative bulk data transfer protocol, in: Proceedings of IEEE Infocom, Hong Kong, March 2004.

[34] Squid Web Site <http://www.squid-cache.org/>.

[35] Ion Stoica, Robert Morris, David Karger, M. Fran Kaashoek, Hari Balakrishnan, Chord: a scalable peer-to-peer lookup service for internet applications, in: Proceedings ACM SIGCOMM'2001, San Diego, CA, August 2001, pp. 149–160.

[36] Marvin Theimer, Michael B. Jones, Overlook: scalable name service on an overlay network, in: Proceedings of the 22nd ICDCS, Vienna, Austria, July 2002.

[37] Limin Wang, Vivek Pai, Larry Peterson, The effectiveness of request redirection on CDN robustness, in: Proceedings of the Fifth OSDI Symposium, Boston, December 2002, pp. 345–360.

[38] Ben Y. Zhao, John Kubiatowicz, Anthony D. Joseph, Tapestry: an Infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California University, Berkeley, Computer Science Division (EECS), April 2001.

Ye Xia is an assistant professor at the Computer and Information Science and Engineering department at the University of Florida, starting in August 2003. He has a Ph.D. degree from the University of California, Berkeley, in 2003, an MS degree in 1995 from Columbia University, and a BA degree in 1993 from Harvard University, all in Electrical Engineering. Between June 1994 and August 1996, he was a member of the technical staff at Bell Laboratories, Lucent Technologies in New Jersey. His research interests are in computer networking area, including performance evaluation of network protocols and algorithms, congestion control, resource allocation, and load-balancing on peer-to-peer networks. He is also interested in probability theory, stochastic processes and queueing theory.

Shigang Chen (sgchen@cise.ufl.edu) received his B.S. degree in computer science from University of Science and Technology of China in 1993. He received M.S. and Ph.D. degrees in computer science from University of Illinois at Urbana-Champaign in 1996 and 1999, respectively. After graduation, he had worked with Cisco Systems for three years before joining University of Florida in 2002. His research interests include network security and wireless networks. He received IEEE Communications Society Best Tutorial Paper Award in 1999 and NSF CAREER Award in 2007. He was a guest editor for ACM/Baltzer Journal of Wireless Networks (WINET) and IEEE Transactions on Vehicle Technologies. He served as a TPC co-chair for the Computer and Network Security Symposium of IEEE IWCCC 2006, a vice TPC chair for IEEE MASS 2005, a vice general chair for QShine 2005, a TPC co-chair for QShine 2004, and a TPC member for many conferences including IEEE ICNP, IEEE INFOCOM, IEEE ICC, IEEE Globecom, etc.

Chunglae Cho received the B.S. and M.S. degrees in computer science from Pusan National University, Korea, in 1994 and 1996, respectively. He worked as a research staff member at Electronics and Telecommunications Research Institute, Korea, between 2000 and 2005. He is currently working toward a Ph.D. degree at the Computer and Information Science and Engineering department at the University of Florida. His research interests are in computer networks, including resource allocation and optimization on peer-to-peer networks and wireless networks.

Vivekanand Korgaonkar completed his master's degree in Telecommunications from University College London in 2007. He also has a master's degree in Mathematics and Computing from the Indian Institute of Technology, Delhi. He is a former research scientist from BT Research Labs where he developed two patents on sensor networks. He is currently working as transformation consultant.