Load Balancing with Multiple Hash Functions in Peer-to-Peer Networks

Ye Xia, Shigang Chen and Vivekanand Korgaonkar
Computer and Information Science and Engineering Department
University of Florida
Gainesville, FL 32611-6120
Email: {yx1, sgchen, vk2}@cise.ufl.edu

Abstract

Peer-to-peer(P2P) networks have grown in popularity in recent years. One of the typical applications of P2P networks is file-sharing. Effective load balancing in such applications is important since the distribution of the number of requests for individual files can be heavily skewed. In the basic design of these networks each file is stored at a single node (i.e., server) which will become a hotspot if the file is popular. In this paper, we focus on the file-replication strategy that utilize multiple hash functions. Such a strategy typically sets 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 server 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 to develop a set of distributed algorithms that implement the above solutions and to evaluate their performance. In particular, we analyze a random binary search algorithm and random gap-removal algorithm.

1. Introduction

Peer-to-Peer (P2P) networks have been popularized by file-sharing networks such as Napster [8], Gnutella [4] and Kazaa [7]. Such a network is typically overlayed on top of the IP network, or the Internet, where each node of the network is typically an ordinary computer and each (virtual) link is an end-to-end IP network path. Virtually all existing P2P networks have no specific requirement on how nodes are connected to each other, and therefore, are said to be *unstructured*. File searching in these networks requires either query flooding or establishing file directories in advance. In the past few years, researchers have proposed several *structured* networks such as CAN [12], Chord [15], Pastry [13], Tapestry [17], and Plaxton-type networks [9]. These networks establish the routing tables at the time of network construction or as nodes join or leave the network, and hence, eliminating the need of a routing protocol. Each network implements a distributed hash table (DHT), distributed in the sense that different pieces of the table are stored separately across different nodes. Publishing a file is to insert the file into the hash table. More specifically, a hash function is first applied to the file and the returned hash value becomes the file ID. The file is then published at a node that owns the 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 and to route a query with the hash value as the destination address. Thus, the combination of hashing and structured routing remove the need of query flooding or establishing file directories.

In this paper, we consider a scenario of data retrieval or file download in a structured P2P network with a large number of users. Large file-sharing P2P networks (i) should have build-in resiliency or fault tolerance to combat the problem of *single point of failure*, that is, when a node fails, files at the node will no longer be accessible; (ii) should have load-balancing mechanism to solve the *hotspot* problem, that is, all requests for a particular file are directed to one node, causing overload to that node or the network paths leading to the node; and (iii) preferably, should replicate the file content to increase the lifetime of each file.

The subject of the paper is on data or file replication techniques using multiple hash functions. The idea 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 publish copies of the file in k nodes. One

Proceedings of the 12th International Conference on Parallel and Distributed Systems (ICPADS'06)
0-7695-2612-8/06 $20.00 © 2006 IEEE

of the challenges is that it is not easy to decide the number of hash functions needed, which is file dependent. The strategy in this paper is to set aside a *large* number, m, hash functions, large enough for the most demanded file (e.g., $m = 2^{32}$). Without loss of generality, suppose each hash function has a unique ID from 1, 2, ..., m. How many of these functions are actually used depends on the popularity of each particular file. We expect that most of them are not used for the majority of the files. The central problems are how a node chooses one of the unused hash functions when replicating a file, or one of the used hash functions when requesting a file. Our solution to the former problem is to choose the first unused hash function for file replication. As a result, when k hash functions are being used, they must have IDs 1, 2, ... k. Our solution to the latter problem is to choose a used hash function uniformly at random. Our main contribution is to develop a set of distributed algorithms that implement the above solutions and to evaluate their performance. In particular, we analyze a random binary search algorithm and random gap-removal algorithm. We stress that using multiple hash functions for load balancing has been proposed and discussed in several previous studies (See Section 1.1.) The contribution of this paper is, therefore, not in re-discovering the idea of using multiple hash functions, but in solving the technical problems related to the use and management of the hash functions.

The mechanisms for hash function usage and management must be efficient and simple, and in addition, must cope with the characteristics of the P2P network we envision. For instance, the network is highly dynamic with nodes freely joining and leaving, node failure may be frequent, and the level of security may be low. In such a highly dynamic network, it is 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. We also make an assumption that the files are much larger than protocol control messages. This implies that each node can process much more request messages than the number of file downloads it can serve per unit of time. We therefore make the distinction between access to a node by the control messages and the actual selection of the node as a downloading server.

1.1. 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, i.e., the nodes to which it has direct (virtual) links, or at 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). Tapestry uses strategy (ii) and (iii). Following the suggestions in Chord, CFS [2] replicates a file at k successors of the original server and also caches the file on the search path. PAST [14], 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 [9], 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.

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.

In [6], [1], and [10], file replication is performed through multiple hash functions, which are organized in a tree. This results in the cache servers being organized into a tree. One drawback of this approach is that

some servers retrieve and copy the file but do not serve it, leading to inefficient use of resources. A more serious problem is that the servers in the tree are not equal. Those at a higher level of the tree have more "power" than those at a lower level. This opens the possibility that one or a small number of malicious users attack and compromise the most important servers in the tree and bring down a large portion of the network.

All aforementioned approaches achieve the load-balancing objective through file replication/caching. There is a complementary approach suggested in [2] and studied in [11] and [5] that monitors the server load as a whole. Instead of replicating the files in a heavily loaded server, some of its files are moved to lightly loaded servers. This can be done by dividing a real server into multiple virtual servers with different IDs. Deleting a virtual server most likely will move its files to other real servers.

2. Basic File Replication and Access Algorithms

2.1. Replication with Multiple Hash Functions

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 $h_1, h_2, ..., h_m$, 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 $i = 1, 2, ..., m$. For instance, if the argument is the file name, *foo*, then $h(foo1), h(foo2), ..., h(foom)$ gives m hash values for the file. The number i is called the "salt" value in [17]. For an in-depth discussion on creating proper hash functions, the readers are referred to [6].

Since m is a very large number, we do not want to replicate every file m times. Instead, we will take the *popularity-based* file-replication strategy, which has been proposed in previous studies such as [6] [16]. 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 overall request rate increases, threatening to overload the servers that currently contain the file, how does the network replicates the file to other severs, with the help of the unused hash functions? This sub-section gives an answer to the second question.

With respect to a fixed file, let us call a node (i.e., server) that already contains a copy of the file a *filled node*. Otherwise, the node is called an *empty node*. In the so-called *push* strategy, file replication is initiated by the 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. We will only discuss the push strategy in this paper.

The goal of our file-replication algorithm is that, if k hash functions are used for replication, they must be $h_1, h_2, ..., h_k$. The rationale for this will become apparent when we discuss how to use the hash functions to access the file in Section 2.2. 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, $h_1, ..., h_k$. 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 $s, s+1, ..., t$, 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 $i = \lfloor (s+t)/2 \rfloor$ [1]. This is accomplished by routing the query with the $h_i(f)$ as the destination address in the P2P network. 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 ways for a node to decide if it is overloaded (with respect to a fixed file). For instance, it can measure the backlog of file requests, i.e., the number of received requests that are being or to be served. Alternatively, it can measure the average request arrival rate over some measurement interval. A filled node can then compare the measured value against some pre-defined threshold, θ, and decide whether it should replicate the file. The measurement-based file replication share many common features with other types of measurement-based control in areas such as congestion control, queue management and admission control, which have all been studied extensively. Even though the ideal operation of the measurement-based control typically requires delicate tuning of the

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

control parameters or more complex adaptive control scheme, we will show in Section 3.1 by simulation that simple schemes often suffice. Finally, if the request rate is below a threshold $\zeta < \theta$, the node removes the file (or simply marks the file as being deleted).

2.2. File Access: 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 $h_1, ..., h_k$ 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 another number between 1 and i_1 randomly, say i_2. If h_{i_2} is again not used, in the next iteration, we pick another 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. Performance Evaluation of the Basic Algorithms

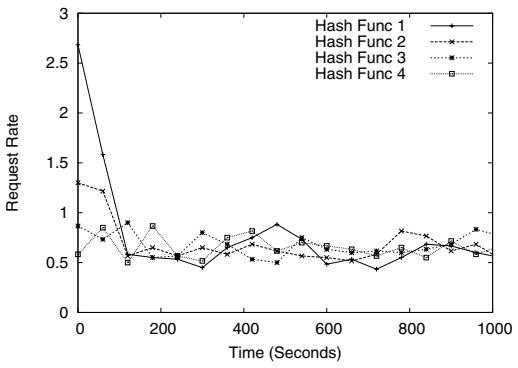
3.1. Simulation Experiments

In the simulation, each filled node measures the average request rate for a fixed file on each measurement interval. Figure 1 (a) and (b) show the request rates measured on 60-second and 10-second intervals, respectively. The total request rate for the file is set at 2.5 per second. If the request rate measured by a filled node exceeds 1.0 per second, the node starts to push a copy of the file to another node. In these simulation runs, each hash function maps the file to a distinct node, hence, each hash function corresponds to a distinct node. For ease of description, we will index the nodes by their corresponding hash function indices. That is, node i corresponds to the hash function h_i. At time 0, the file resides only at the node 1, and the node experiences a high request rate. As the file-replication algorithm proceeds, copies of the file are pushed to other nodes exponentially fast. For instance, for the case with 60-second measurement interval, the file is pushed by node 1 to node 2 at time around 60 second. At time around 120 second, it is pushed by node 1 to node 3 and by node 2 to node 4. Eventually, each of the four filled nodes experiences the same long-run request rate of about 0.64 per second. Comparing Figure 1 (a) and (b), in the case with 10-second measurement interval, the request rate measured by each node has much larger fluctuation than in the case with 60-second measurement interval, due to smaller number of samples collected on a smaller time interval in the former case. In the 10-second case, the file is eventually pushed to 7 other nodes, as compared to 3 other nodes in the 60-second case. Each of the 8 nodes sees a long-run request rate around 0.3 per second.

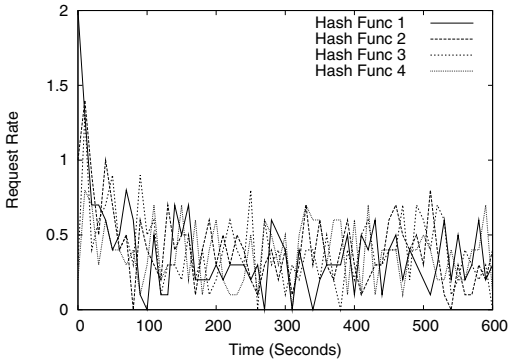
The simulation results for file replication presented here are brief because they are as expected and are not the main focus of the paper. We also mention in passing that, in the more complicated situation with many-to-one mapping from the hash functions to the nodes, the request rate seen by each filled node is proportional to the number of hash functions the node corresponds to. It may be desirable that the replication threshold is scaled properly with the number of hash functions. This prevents the situation that some nodes are replicating the file and others are not and that replication stops only when the most loaded node experiences a request rate lower than the threshold value. Alternatively, the pull strategy for replication also prevents the above scenario from occurring.

3.2. Analysis on the Random Binary Search Algorithm

3.2.1. Hash function selection. Let $T(m)$ be the number of steps taken before a used hash function is returned. 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.



(a)



(b)

Figure 1. Request rate seen by filled servers. (a) measurement interval = 60 seconds; (b) measurement interval = 10 seconds

In addition, 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. For brevity, we omit the proofs.

Theorem 3.1

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

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

$$1 + \ln \frac{m}{k} \leq \mathbf{E}T(m) \leq 1 + \ln \frac{m-1}{k-1}. \quad (2)$$

Let $\mathrm{Var}(X)$ denote the variance of the random variable X. We can show

Theorem 3.2 *For $m > k$,*

$$Var(T(m)) = \frac{1}{k^2} + \frac{1}{(k+1)^2} + ... + \frac{1}{(m-1)^2} + \frac{1}{k} + \frac{1}{k+1} + ... + \frac{1}{m-1}. \quad (3)$$

For $1 < k < m$, reasonable bounds for $\mathrm{Var}(T(m))$ are

$$\ln \frac{m}{k} + \frac{1}{k} - \frac{1}{m} \leq \mathrm{Var}(T(m)) \leq \ln \frac{m-1}{k-1} + \frac{1}{k-1} - \frac{1}{m-1}. \quad (4)$$

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

3.2.2. Access to all hash functions. In the load-balancing application on P2P networks, we wish to load-balance the file servers by choosing one of them uniformly for downloading. We also wish not to overload other nodes corresponding to the unused hash functions with excessive query traffic. We have just established that each used hash function is selected with equal probability. However, the access pattern to the unused hash functions by 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 $k < i \leq m$.

To answer this question, we work with a continuous 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 $0 < a \leq 1$. 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. Let the random variable T be the number of steps taken before the algorithm returns some $y \in [0,a]$. It is possible to compute the distribution of T and its first and second order statistics. Let Z_i be the position of the i^{th} jump in the algorithm, $i = 1,2,\ldots$. Let us consider the stopped process, $Z_1, Z_2, ..., Z_T$. For each $0 \leq y \leq 1$, let $N(y)$ be the number of Z_i's less than or equal to y in the stopped process. That is

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

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

Theorem 3.3

$$n(y) = \begin{cases} \frac{1}{a} & \text{for } 0 \leq y \leq a \\ \frac{1}{y} & \text{for } a < y \leq 1 \end{cases}. \quad (5)$$

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 3.3 should also approximately apply to the discrete case. In Figure 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 $1 \leq n \leq m$ and $1/k$. We see that Theorem 3.3 applies very well here.

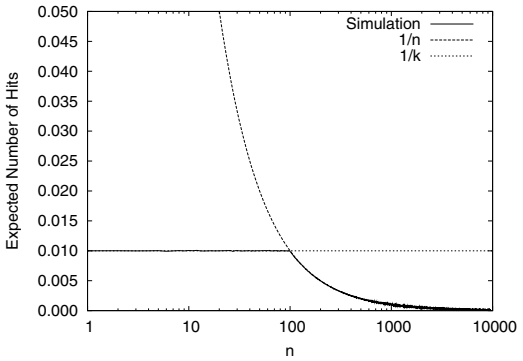


Figure 2. Expected number of hits to each hash function, for $m = 10000$ and $k = 100$.

4. The Compacting Problem

4.1. Motivation

The hash function usage scheme presented in Section 2 is adequate if no node ever leaves the P2P network unexpectedly. If a node leaves the network without running the proper protocol for leaving, the files it contains will not be moved to appropriate neighboring nodes to ensure their continued availability. From the point of view of the hash functions, unexpected node departure creates gaps in the sequence of used hash functions. Without proper repair, the number of 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 no longer in use. 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.

4.2. Gap Removal: The Compacting Scheme

We will consider the following simple gap removal algorithm. Let us focus on a particular file, say, f. For ease of discussion, let us also assume each hash function corresponds to a distinct node, and hence, the relevant nodes $h_1(f)$, $h_2(f)$, ..., $h_m(f)$ are all different. Every once in a while, each filled node randomly checks an earlier node to see if the corresponding hash function is in use. If not, that function will be put to use and the hash function corresponding to the checking node is removed from usage. More specifically, the filled node $h_j(f)$ draws a number I randomly from $1, 2, ..., j-1$, then sends a query message for f to $h_I(f)$. If h_I is no longer in use, indicated by the fact that node $h_I(f)$ does not contain f, then a replica of f is created at node $h_I(f)$ and the replica at $h_j(f)$ is removed. 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 Figure 3, where h_6 is removed and the gap at h_2 is filled.

Let us now specify how often a filled node should attempt such a check. For every filled node, let the interval between two consecutive checks be an exponential random variable with mean $1/\lambda$, and let the checks by different nodes be independent from each other. The sequence of checks form a continuous-time Markov chain, where the checks occur at a rate $k\lambda$, where k is the number of used hash functions. We will focus on the embedded discrete-time Markov chain. In the following we will try to understand some properties of this algorithm.

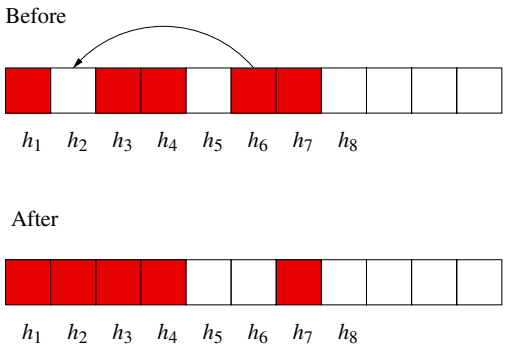


Figure 3. Gap removal: remove h_6 and use h_2

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 $1 \leq k \leq m$. Each 1 corre-

sponds to a used hash function, and each 0 corresponds to an unused function. In accordance with the objective of the algorithm outlined above, 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 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 $i \in \{1,2,...,m\}$, 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 $1 \le j < i$ and $g_i(j)$ satisfies $\sum_{j=1}^{i-1} g_i(j) = 1$ for each i. If the j^{th} position to the left of position i is a 0, then moving the 1 is allowed. In other words, the 1 at position i and the 0 at position $i-j$ exchange positions. Otherwise, the 1 is not moved. We wish to know how long it takes to compact the vector x.

The sequence of transitions form a finite-state Markov chain, denoted by $\{X^n\}_{n=0}^{\infty}$. For any two vectors $x, y \in \{0,1\}^m$, we write $x \rightsquigarrow y$ if y can be derived from x by moving a 1 in x to a position to its left that contains a 0. It must be true that x and y are identical at all positions except at two positions i and j, where $i < j$ and $x_i = 0$, $y_i = 1$, $x_j = 1$, and $y_j = 0$. For instance, if $x = 1010$ and $y = 1100$, then $x \rightsquigarrow y$. Since such positions i and j depend on x and y, we rewrite i as $L(x,y)$ and j as $H(x,y)$. Let $S(x) = \{y \in \{0,1\}^m : x \rightsquigarrow y\}$, the set of vectors derived from x by exchange a 1 with a 0 to the left. With these, the transition probabilities of the Markov chain, denoted by $p(x,y)$, are given by

$$\begin{aligned} & p(x,y) \\ =& P\{X_{n+1} = y \mid X_n = x\} \\ =& \begin{cases} \frac{1}{k} g_{H(x,y)}(H(x,y) - L(x,y)) & \text{if } x \rightsquigarrow y \\ 1 - \sum_{z \in S(x)} p(x,z) & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}. \end{aligned}$$

Given the Markov chain starts at $X_0 = x$, the time to finish compacting x is denoted by T_x. Given fixed m and k, the expected value of T_x, denoted by $\mathbf{E}T_x$, can be computed for all vector x of length m with exactly k 1's. This is done as follows. First, let us interpret each vector x as a binary representation of a non-negative integer and order all vectors of length m and with k 1's in decreasing order of the numerical value. There are exactly $\mathbf{C}_k^m$ of such vectors. This is the number of different ways to place k 1's at m positions. The ordered vectors are denoted $x^1, x^2, ..., x^{\mathbf{C}_k^m}$. We write $v(i) = \mathbf{E}T_{x^i}$. We also interpret $p(i,j)$ as $p(x^i, x^j)$ in the original notation.

Starting with $X_0 = x^i$ and conditional on the first jump, for $1 < i \le \mathbf{C}_k^m$,

$$v(i) = \sum_j p(i,j)v(j) + 1. \tag{6}$$

Also,

$$v(1) = 0. \tag{7}$$

We rewrite (6) and (7) in matrix notation.

$$v = Pv + r, \tag{8}$$

where $v = (v(1), v(2), v(\mathbf{C}_k^m))^T$, P is the transition matrix of the Markov chain, and $r = (0,1,1,...,1)^T$. From Lemma 2 in chapter 4 (page 123) of [3], the solution to (8) exists and is unique. For our problem, the transition matrix, P, is lower-triangular. The solution to (8) can be found easily by computing $v(1), v(2), ...$ iteratively. The difficulty lies in the potentially large dimension of the vector v for large value of m. We will consider some special cases of (8).

4.3. Uniform Jump

Imagine that the 1 in the j^{th} location of the vector x is picked to make a jump, where $1 < j \le m$, and the position to jump to is chosen uniformly at random on $[1, j-1]$. In other words, $g_j(l) = \frac{1}{j-1}$, for all $l = 1,2,...,j-1$. This is called the *uniform jump* algorithm.

4.3.1. 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 0's. An example is 1111000100 for i=3. Let us re-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, $v(0) = 0$. We can show that

Lemma 4.1

$$v(i) = \begin{cases} k^2 & i = 1 \\ k^2 + k\sum_{j=2}^{i} \frac{1}{j} & 1 < i \le m-k \end{cases}. \tag{9}$$

4.3.2. 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 0's. In other words, it has the form 1...101...10...0. Let us re-index the vectors so that the i^{th} 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 i^{th} vector, the isolated 0 is followed by i consecutive 1's. For convenience, let us call the vector 1...10...0 the 0^{th} vector, and let $v(0) = 0$. We can show that

Lemma 4.2 *For $i = 1,2,...,k$, $v(i) = k^2$.*

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 4.1 and 4.2, since the number of jumps per unit time scales linearly with k. 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 (9), 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 4.1 and 4.2. 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.

4.4. Simulation Experiments for Compacting the Bit Array

Let us call the bit vector as a bit array for convenience. In all subsequent simulation results, the time unit is normalized in the following way. Each array 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 time intervals 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 marked entries, i.e., the number of 1's in the array.

We consider the following class of randomized compacting algorithm, called `compact(p)`, independently run by each marked entry. Let us focus on a marked entry at location (index) j, where j ranges from 2 to m. With probability p, it picks the target location $t = j-1$ to make a jump attempt, and with probability $1-p$, it picks a target t uniformly at random from $[1, j-1]$. If the target location t contains a value 1, then nothing is done. If it contains a 0, then the marked entry at location j is moved to location t. In other words, the value at location t becomes 1 and the value at location j becomes 0. The special case of `compact(0)` is the uniform jump algorithm. 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*. An array of this type 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", in suitably defined sense, for many compacting algorithms, such as uniform jump. However, we have not proven any of these types of claims. 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 $\{1, 2, ..., m\}$ without replacement. This is shown in Figure 4. Note the linear dependence of the mean completion time on k, the number of marked entries, which can be too slow when k is large. This problem will be addressed 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.

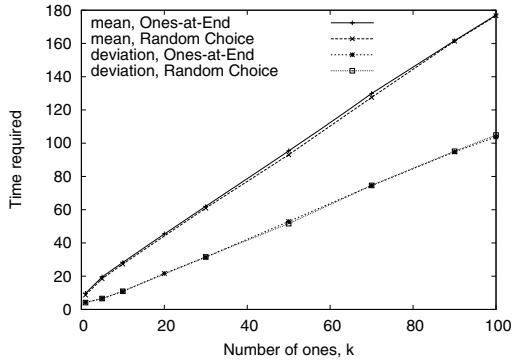


Figure 4. Time required to compact the 1's. Comparison of Ones-at-End and Random-Choice. $m = 10000$

Recall that our objectives for compacting the array of 0-1 digits 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 at location 1 becomes 1. In Figure 5, 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. (See Figure 4. The mean time for the case $k = 1000$ is derived by linear extrapolation of the curve.).

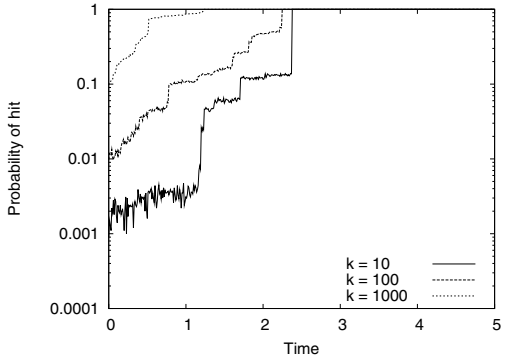


Figure 5. Probability of eventually hitting a marked entry. The initial array is of the Ones-at-End type. $m = 10000$**.**

To examine how well our second objective of the compacting algorithm is fulfilled, in Figure 6, we plot the load to each of the marked entries as the time progresses for the same instances as in Figure 5. 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 Figure 5, the probability of an eventual hit to any marked entry is low. We point out that the load at time 0 is generally not uniform for most initial array patterns. In fact, it is uniform only for arrays in which the marked entries are packed consecutively. 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. To make it more concrete, starting with an initial array such as `0000000000111111`, very quickly, the array is compacted into something like `1111101000000000`. However, it takes much longer time to eventually finish the compacting process. This can be confirmed by observing that, starting with an array of the Isolated-1 type with $m = 10000$, $k = 100$ and $i = 1$, where i is the number of 0's before the last isolated 1, it takes 100 time units to complete the compacting process (See Lemma 4.1.), whereas the mean completion time starting from an array of the type Ones-at-End is only 177.12.

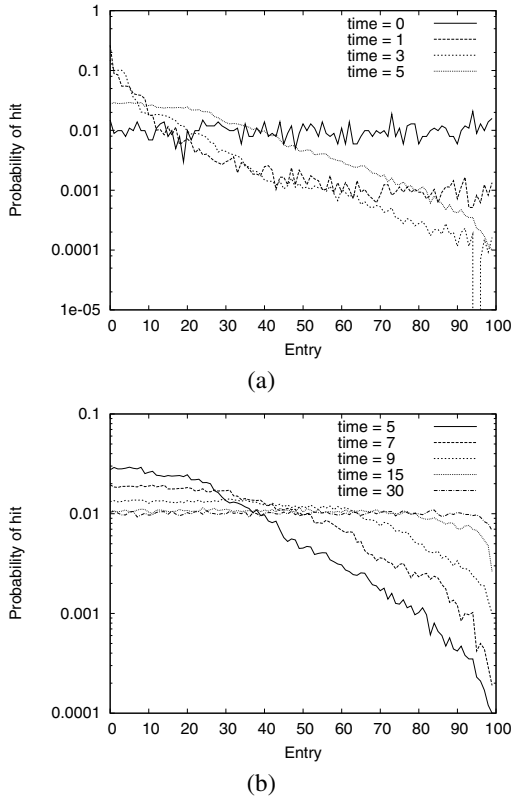


Figure 6. Load to the marked entries over time. The initial array is of the Ones-at-End type. $m = 10000$**,** $k = 100$**. (a) during time 0 to 5; (b) during time 5 to 30**

5. Conclusion

This paper deals with algorithmic issues related to the file replication strategy using multiple hash functions for structured P2P networks. The central issue here is, out of the potentially large number of hash functions, which one to use for downloading or which one to use for replicating a file so that the load to the servers is balanced. Our main contribution is that, first, we have devised the random binary search algorithm and the associated hash function compacting scheme. These algorithms are suitable to the dynamic P2P environment and are potentially more tolerant to security breaches than previously known tree-based approaches for organizing the hash functions. Second, we have explored the performance of these algorithms.

Throughout the paper, we have been focusing on a single file. In reality, since a server can serve multiple files, balanced load with respect to one file does not imply balanced load with respect to all files. We feel that it is not an urgent priority to study this because, in the P2P environment, each host most likely contains a small number of files, and the chance is small that more than one files on it are popular simultaneously. More importantly, in addition to the basic load-balancing mechanisms suggested in the paper, each server should also implement an admission control mechanism so that it can reject new requests if it is overloaded, irrespective of which files have caused it. Admission control is also a solution to some other problems that have been overlooked so far, such as many-to-one mapping from the hash functions to the servers and heterogeneous server capacities.

References

[1] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael Schwartz, and Kurt Worrell. A Hierarchical Internet Object Cache. In *Proceedings of USENIX Annual Technical Conference (USENIX '96)*, San Diego, CA, January 1996.

[2] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP '01)*, pages 202–215, Banff, Alberta, Canada, October 2001.

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

[4] Gnutella Forums Website. `http://www.gnutellaforums.com`.

[5] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load Balancing in Dynamic Structured P2P Systems. In *Proceedings of IEEE Infocom 2004*, Hong Kong, March 2004.

[6] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing (STOC '97)*, El Paso, TX, May 1997.

[7] Kazaa Website. `http://www.kazaa.com`.

[8] Napster. `http://www.napster.com`.

[9] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 311–320, Newport, Rhod Island, June 1997.

[10] C. Greg Plaxton and Rajmohan Rajaraman. Fast Fault-Tolerant Concurrent Access to Shared Objects. In *Proceedings of the twenth-eighth Annual ACM Symposium on Theory of Computing (STOC '96)*, Philadelphia, PA, May 1996.

[11] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, pages 311–320, Berkeley, CA, Feb. 2003.

[12] Sylvia Ratnasamy, Paul Francis, Mark Hanley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM '2001*, pages 161–172, San Diego, CA, August 2001.

[13] Antony Rowstron and 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.

[14] Antony Rowstron and 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)*, pages 188–201, Banff, Alberta, Canada, Oct 2001.

[15] Ion Stoica, Robert Morris, David Karger, M. Fran Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM '2001*, pages 149–160, San Diego, CA, August 2001.

[16] Marvin Theimer and Michael B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the 22nd ICDCS*, Vienna, Austria, July 2002.

[17] Ben Y. Zhao, John Kubiatowicz, and 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.