

# Multiple-Choice Random Network for Server Load Balancing

Ye Xia, Alin Dobra and Sueng Chul Han  
Computer and Information Science and Engineering Department  
University of Florida  
Gainesville, FL 32611-6120  
Email: {yx1, adobra, schan}@cise.ufl.edu

**Abstract**—In many networking applications such as file sharing, structured peer-to-peer networks are increasingly used in dynamic situations with fluctuating load, which require proper load balancing. The relationship between the network structure and its load-balancing properties has not been fully understood. In this paper, we focus on the Plaxton-type networks, which are broad enough to include Pastry, Tapestry, and hypercube. We first use hypercube as an example and demonstrate that replicating files at nodes in decreasing order of the length of the common prefix with the original server leads to perfectly balanced load, and does so fast and efficiently. Moreover, this replication strategy coincides with a simple on-demand replication/caching strategy based on the observed load. One of our main contributions is to show that such desirable properties also exist for a large class of random networks, which are less restrictive and more practical than the hypercube. More importantly, we have discovered a multiple-choice random network, which drastically reduces the statistical fluctuation of the load: The maximum load over all replication servers is at most three times the average load for systems of practical sizes. The main insight is that this algorithm is related to a variant of the multiple-choice balls-in-bins problem.

## I. INTRODUCTION

### A. Motivation

Structured networks have recently attracted much attention in the areas of peer-to-peer (P2P) networks or other overlay networks due to their desirable characteristics, such as enabling large-scale resource sharing, decentralization of massive computation, simplified routing, fast resource location (e.g., due to small network diameter), fault-tolerance due to path redundancy, scalability, and ease of management. They have been applied to diverse applications such as application-level multicast, persistent data storage, web related service, file sharing, media streaming and content distribution. Among recently proposed systems, as pointed out in [1], Chord [2], Tapestry [3], and Pastry [4] are related to the hypercube topology, CAN [5] is based on the torus topology, Koorde [6] and ODRI [7] are based on the de Bruijn graph, Viceroy [8] and Ulysses [9] are based on the Butterfly topology, and FISSIONE is based on Kautz graphs [1].

Most structured networks are invented for their efficient self-routing property. That is, the network is constructed with a routing structure so that a message can be routed based on its destination ID. This is in contrast to networks that need to dynamically update routing tables using a routing protocol, such as the Internet. Moreover, most of the structure networks have small network diameters. Increasingly, structured networks are applied to file/data distribution, storage, access and sharing, which face fluctuating load and require proper load balancing.

The relationship between the network structure and its load-balancing properties has not been fully understood. The main questions are: What type of network structure is convenient for load balancing? Given a particular network structure, what is the distribution of the load over the network nodes and what are suitable load-balancing techniques? If files should be replicated, where should we place the replicas?

This paper represents a step in the systematic study of such questions. The main contribution is that we have discovered a random, Plaxton-type network that has the self-routing property and a small network diameter, but in addition, has desirable load-balancing features. The Plaxton-type networks are broad enough to include Pastry, Tapestry and hypercube, and are fundamentally related to Chord [10]. We show that it is important to engineer the network structure with load balancing in mind, because, for some networks, transparent load balancing is impossible. By transparent load balancing, we mean that the requesting node is unaware that load balancing is taking place. For those networks where transparent load balancing is possible, the specific load-balancing techniques lead to drastically different performance. For instance, intuitive solutions such as replication at neighbors do not necessarily lead to balanced load, and can waste the resources of under-loaded servers.

Our solution is to construct a Plaxton-type random network through a multiple-choice algorithm and to adopt a replication strategy that replicates the popular file at nodes in decreasing order of the length of the common prefix with the original node (server) (known as LCP-replication). This leads to perfectly balanced load *in the average sense*, and does so fast and efficiently. Moreover, this replication strategy coincides with a simple on-demand replication/caching strategy based on the observed load. Due to the multiple-choice part, the strategy drastically reduces the statistical fluctuation of the load across the servers. The *maximum load* over all replication servers (a.k.a *cache servers* or *cache nodes*) is at most three times the *average load* for systems of practical sizes.

The intellectual motivation of the above design comes from two areas. The first is LCP-replication on the hypercube network, which leads to perfectly balanced load quickly and efficiently. However, such a network can be restrictive for practical purpose. For instance, the ID space must be fully occupied by nodes. For more practical networks, we first show that the same conclusion holds for a *single-choice* Plaxton-type random network, when the load is averaged over many random network instances. For a fixed network instance, the maximum load can be up to ten times the average load across all cache

servers. Such results come from the area of study about sending  $m$  balls into  $n$  bins by choosing the bin randomly for each ball, known as the balls-in-bins (BNB) problem.

Another of our contributions is to make the connection between a recursive BNB problem with the random network construction problem. The utility of making such a connection lies in that, a direct simulation study cannot handle a network with more than several tens of thousands of nodes, while one can simulate the BNB problem with up to  $2^{26}$  balls. In addition, the known theoretical results about the BNB problem allow us to interpret the simulation results with confidence and to extrapolate them for even large network size. In this paper, we have made performance-related conclusions about networks with  $2^{64}$  nodes. Perhaps more importantly, motivated by the multiple-choice BNB problem, we are able to invent the multiple-choice random network construction algorithm, which drastically reduces the maximum load for any fixed instance of the random network.

We will use file access on P2P or other overlay networks as the main application scenario, since it is fairly generic. Effectively load balancing is extremely important in large-scale file-access applications since the popularity of files often exhibits a heavy tail distribution, and hence, the servers for popular files can be many times overloaded. We assume that the files themselves, instead of just the pointers to them, need to be replicated. We also assume that the files are large so that, when the demand is high, large-scale replication is necessary and incurs non-trivial cost, and hence, should be optimized. Every node and file has an ID, obtained by applying a uniform hash function [11] to certain attributes of the node or the file. The resulting IDs are distributed uniformly in a common ID space. *As a result, the origins of the queries for a file are uniformly distributed across the network, which is a key assumption of the paper.* A published file is stored in a node whose ID is “close” to the file ID, in the sense specified by the network designer.

The rest of the paper is organized as follows. In Section II, we introduce the LCP-replication scheme on the hypercube. This serves to illustrate the relationship between the load-balancing technique and the network structure and motivates the construction of a more applicable random network. The multiple-choice random network is presented in Section III. In Section IV, we relate the recursive BNB problem to the random network construction algorithm and present a detailed study of LCP-replication on large networks. We conclude in Section V.

### B. Past Work

File-replication strategies relevant to our work can be summarized into three categories, which are complementary rather than mutually exclusive: (i) caching, (ii) replication at neighbors or nearby nodes, and (iii) replication with multiple hash functions. The essential difference among them lies in where the replicas of the file is placed. In (i), the 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 (ii), when a node is overloaded, 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 [12] replicates a file at  $k$  successors of the original server and also caches the file on the search path. PAST [13], 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 [14], 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. In [11], file replication is performed, in essence, through multiple hash functions. Hash-function-based replication has its own technical challenges, such as hash function management [15], and will not be considered in this paper. Several other works on load balancing are more distantly related to our work, including [16], [17], [18], [19], [20], [12], [21], [22].

The load-balancing performance of strategy (i) and (ii) is topology-dependent and this dependency has not been systematically studied. Replica-placement and network structure are jointly studied in [14] and [23]. However, the objective there is to reduce the access delay rather than server load balancing. The focus of our work is to discover different structured networks so that some simple replica-placement strategy delivers good load-balancing performance.

## II. DETERMINISTIC NETWORK: PERFECT LOAD-BALANCING ON HYPERCUBE

In this section, we define the structure and routing of the Plaxton-type networks and the hypercube. We describe LCP-replication and LCP-caching on the hypercube, which lead to perfectly balanced load at a fast speed, are easy to use, and are robust. The same load-balancing techniques will be used for the multiple-choice random network with similar performance. However, the intuitive justification is best seen on the hypercube.

### A. Structure and Routing of Plaxton-Type Networks

The class of Plaxton-type networks have the prefix-based self-routing property and the path length is at most  $\log_2 m$ , where  $m$  is the size of the name (i.e., ID) space. Consider a network with  $n$  nodes,  $n \leq m$ , taking IDs from the name space  $0$  to  $m - 1$ . Each file is mapped into a key value in the same name space  $\{0, 1, \dots, m - 1\}$  through a uniform hash function, and is stored at the node whose ID is the “closest” to the file key. There is some flexibility in the definition of *closeness* [19]. Suppose  $2^{e-1} < m \leq 2^e$ , where  $e$  is a natural number. The node IDs and the file keys can all be expressed as binary numbers.

When a file is requested, the query message is routed from the requesting node, to the node that contains the file. Suppose the requesting node ID is  $a_{e-1}a_{e-2} \dots a_0$  and the file key is  $b_{e-1}b_{e-2} \dots b_0$ , where  $a_i, b_i \in \{0, 1\}$  for each  $i$ . In a Plaxton network, routing of the query from node  $a_{e-1}a_{e-2} \dots a_0$  to node  $b_{e-1}b_{e-2} \dots b_0$  can be viewed as converting the former number to the latter, one digit at each hop from the left to the right. More specifically, by moving to the  $i^{th}$  hop,  $i = 1, 2, \dots, e$ ,  $a_{e-i}$  is changed into  $b_{e-i}$  if they are not the same. This is known as *prefix routing*. As an example, suppose  $m = n = 2^5$ , and suppose node 10110 makes a query for file 00000, which resides in node 00000. A possible route in a Plaxton network may consist of the following sequence of

nodes: 10110  $\rightarrow$  00101  $\rightarrow$  00101  $\rightarrow$  00011  $\rightarrow$  00001  $\rightarrow$  00000. In practice, the repeated node, 00101, shows up on the path only once.

The key to the above routing behavior in a Plaxton-type network is the choice of the routing tables. In this paper, we define the Plaxton-type network as one whose routing table at every node has the form shown in Table I. Table I shows the routing table of node  $a_{e-1}a_{e-2}\dots a_0$ . The search key is checked against the routing table when making the routing decision. The first column is called the *level*. During a route lookup, if the query message has traversed  $i - 1$  hops, then the  $i^{\text{th}}$  digit of the search key, counting from the left to the right, is checked against the first column of the routing table. The second column is the value of the digit. The third column is the ID of the next-hop node, i.e., a downstream neighbor. At each level, say level  $i$ , each next-hop node must satisfy the following requirement: (i) the  $i^{\text{th}}$  digit of the next-hop node must match the value in the second column; and (ii) the  $(i-1)$ -digit prefix of the next-hop node must match the  $(i-1)$ -digit prefix of the current node. Note that, at each level and for each value of the digit, there are possibly more than one next-hop nodes satisfying the above rules, all are called *eligible* next-hop nodes or eligible neighbors. Each “\*” in the table is a wild card, which can be either 0 or 1.

The routing rule can be restated in a simpler form. At each node  $s$  and for the query destination (i.e., file key)  $t$ , the first bit position that  $s$  and  $t$  differs determines the level. For the next-hop node, choose the entry corresponding to the value of  $t$  at the  $i^{\text{th}}$  bit position. For instance, if  $s = 00110$  and  $t = 00000$ , the routing table lookup will check level 3 and value 0. This yields the next-hop node of the form 000\*\*.

TABLE I  
GENERAL ROUTING TABLE FOR NODE  $a_{e-1}a_{e-2}\dots a_0$

level/digit	value	next hop
1	0	0 * ...*
	1	1 * ...*
2	0	$a_{e-1}0 * \dots*$
	1	$a_{e-1}1 * \dots*$
...	...	...
e-1	0	$a_{e-1}a_{e-2}\dots a_20*$
	1	$a_{e-1}a_{e-2}\dots a_21*$
e	0	$a_{e-1}a_{e-2}\dots a_10$
	1	$a_{e-1}a_{e-2}\dots a_11$

Each particular choice for the wild cards in the routing table defines a specific version of the Plaxton-type networks, which all have the distinguishing prefix routing behavior, but other than that, may be very different networks with distinct properties. For each routing table entry, the original Plaxton network [14] chooses a next-hop node that is the closest eligible node to the current node, where the distance may be measured in the round-trip time (RTT). The objective is to minimize the file access delay. Tapestry tries to accomplish the same using incremental, distributed algorithms.

### B. Structure of the Hypercube

Suppose  $m = n = 2^e$ . If we let the wild cards in Table I take the same values as the current node ID at the corresponding digits, we get a hypercube. The result is that each next-hop node matches the current node at all but at most one bit

positions. The location of the special bit that can potentially be different coincides with the level of the routing table entry.

A helpful device for visualizing the hypercube is the embedded tree rooted at node 0, as shown in Figure 1, which is known as a binomial tree [24]. The tree paths to the root are allowed paths by prefix routing. The embedded binomial tree rooted at nodes other than 0, say  $i$ , can be derived from Figure 1 by XOR-ing each node ID with  $i$ . A key observation about the binomial tree is that it is imbalanced. For each fixed node, the sizes of the subtrees rooted at each of its children, when ordered increasingly, are  $2^0, 2^1, 2^2, \dots$ . This property has important consequences on the load-balancing techniques and their performance.

### C. Transparent Load Balancing

In some Plaxton-type networks, transparent load balancing is not possible. Suppose, for each routing table entry in Table I, we choose the largest node (in node ID) among all eligible nodes. Then, the two next-hop nodes in level  $i$  are  $a_{e-1}\dots a_{e-i+1}01\dots 1$  and  $a_{e-1}\dots a_{e-i+1}11\dots 1$ . The result is that, no matter where the query for 00...0 originates, the routing takes the query to node 01...1 in one hop. It is not hard to see that the sequence of nodes traversed by any query is  $01\dots 1 \rightarrow 001\dots 1 \rightarrow \dots \rightarrow 0\dots 01 \rightarrow 0\dots 0$ . Hence, there is no load balancing at all in this network regardless where the file is replicated. The example suggests that, to achieve properly balanced load, one must be very careful in constructing the routing tables, and hence, the structured network itself. In Section II-D and II-E, we will show that transparent load balancing is possible in a hypercube and there exists a very simple and robust replica-placement algorithm that leads to perfectly balanced load.

### D. LCP-replication in Hypercube

Upon close examination, a node that shares longer prefix with the search key receives more requests than a node that shares shorter prefix. Given a node  $s$  and a search key  $t$ ,  $s, t \in \{0, 1, \dots, n-1\}$ , let  $l(s, t)$  be the length of the longest common prefix between  $s$  and  $t$ . It is easy to show the following lemma.

*Lemma 1:* Suppose a query for  $t$  originates from a random node, uniformly distributed on  $\{0, 1, \dots, n-1\}$ . The probability that the query passes through node  $s$  on the way to  $t$  is  $\frac{1}{2^{e-l(s,t)}}$ .

One corollary of the lemma is that, given two node  $s_1, s_2 \in \{0, 1, \dots, n-1\}$  with the property  $l(s_1, t) > l(s_2, t)$ , node  $s_1$  sees more queries on their way to  $t$  than  $s_2$  does, if the queries are generated independently from each other and uniformly across all nodes. If we wish to reduce the load to node  $t$ ,  $s_1$  is the preferred location for file replication. This suggests the so-called *Longest-Common-Prefix-Based Replication* (LCP-replication), as shown in Algorithm 1. This algorithm can either be run by the original cache server in a centralized fashion or by each cache node distributedly.

---

#### Algorithm 1 LCP-replication

---

Replicate the file at nodes in decreasing order of  $l(s, t)$ .

---

The LCP-replication algorithm is also proposed in [23] for networks with the prefix-routing property. The concern there is to replicate files at the appropriate “prefix level” based on their popularity so that the average access time is constant.

The load-balancing performance of LCP-replication and its relationship with the network routing structure have not been studied. For brevity, we state without a proof a key property of LCP-replication on the hypercube.

*Lemma 2:* Suppose independent queries for  $t$  originates from random nodes, uniformly distributed on  $\{0, 1, \dots, n-1\}$ . After  $p$  replication steps,  $p = 1, 2, \dots, e$ , the load to each cache node is  $\frac{1}{2^p}$  of the total requests.

Lemma 2 shows that LCP-replication has the nice properties that (i) in each step of replication, the number of nodes that contain the file is doubled, (ii) after each step of replication, the load to each original cache node is reduced by half, and (iii) the final loads to the cache nodes are all identical. Hence, LCP-replication leads to perfectly balanced load quickly and efficiently. Efficiency comes from the fact that, for serving the same number of requests, LCP-replication results in the smallest number of cache servers. This also leads to the least amount of replication traffic, and hence, is bandwidth-saving.

This process is illustrated in Figure 1, where the cache nodes are shaded. The directed edge indicates the source and destination of the replication process. The numerical label indicates the step at which the file is replicated to the corresponding node. At each replication step, a source node copies the file to only one other node. The end result is that each cache node handles exact 1/8 of the query load. In contrast, if we place replicas at all neighbors of the overloaded nodes, much more nodes become cache servers. This leads to increased traffic volume. Under the same bandwidth, it increases the time required for the replication process to complete, and hence, increases the congestion duration at the current servers. After the replication process completes, the most loaded cache nodes each handle 1/8 of all the queries, but some cache nodes, e.g., node 00001, do not handle any.

### E. Automatic Load Balancing by Caching

With active file replication, an overloaded server “pushes” a copy of the file to other nodes. In contrast, file caching is typically a “pull” strategy, where a node automatically retrieves and saves a copy of the file upon seeing many requests. These are complementary approaches for rapid, scalable, and reliable load balancing. LCP-replication has a natural counterpart: an on-demand caching algorithm that automatically achieves load balancing in the spirit of LCP-replication.

Suppose, for each file  $f$ , each node keeps a load threshold,  $\theta_f$ . The node measures the rate of requests for file  $f$ , denoted by  $r(f)$ . The caching algorithm is listed in Algorithm 2, known as *LCP-caching*. The parameter  $\epsilon \geq 0$  is a safety margin that controls the oscillation of the algorithm.

---

#### Algorithm 2 LCP-caching

---

```

if  $r(f) > \frac{1}{2}\theta_f + \epsilon$  then
  The node caches a copy of  $f$ .
else if  $r(f) \leq \frac{1}{2}\theta_f - \epsilon$  then
  The node removes  $f$  (or marks it as being absent).
end if

```

---

There is no novelty in the LCP-caching algorithm. It is similar to many on-demand caching algorithms based on the observed request load at the current node, for instance, [11] and [12]. However, caching (even LCP-replication) in

general does not necessarily work at all in some Plaxton networks, as shown in the example of section II-C. The key is that hypercube possesses the nice property that the simple, distributed, on-demand caching algorithm works very well: It mimics LCP-replication on the hypercube and has the same optimal load-balancing performance. We state without a proof the following lemma about LCP-caching.

*Lemma 3:* Given any initial set of nodes that cache the file, LCP-caching (with  $\epsilon = 0$ ) eventually converges to the LCP-replication outcome.

## III. LCP-REPLICATION IN RANDOMIZED PLAXTON NETWORK

The hypercube requires  $m = n = 2^e$ . While this can be satisfied in managed networks, in large public networks, the nodes may only sparsely populate the name space. We will handle this situation by constructing a random network, which has the desired load-balancing properties as the hypercube.

### A. Single-Choice Randomized Routing Tables

Starting with the generic routing table as shown in Table I, for each entry, we choose a node uniformly at random from all eligible nodes for the entry. For instance, suppose the current node is  $a_{e-1}a_{e-2} \dots a_0$  and  $a_{e-2} = 1$ . The next-hop node at level 2 for digit value 0 is chosen uniformly at random from all *available* nodes of the form  $a_{e-1}0* \dots *$ . The entry at level 2 for digit value 1 can be ignored since route lookup will never use that entry. The resulting network is an instance of a random network. We can show that, whether or not the name space is fully populated by nodes, the random network has the desirable load-balancing properties similar to the hypercube. Without loss of generality, consider the queries for file 0.

*Theorem 4:* Suppose the file is replicated at all nodes  $0\dots 0* \dots *$  with  $p$  wild cards,  $0 \leq p \leq e$ . Suppose a query originates from a node chosen uniformly at random among all nodes. It is equally likely to be served by any of the cache nodes.

We now present simulation results to show the effectiveness of LCP-replication on the random network. Let the name space size  $m = 5000$ . Suppose the network has 500 nodes sparsely populating the name space. This type of situations is the main reason for moving from the hypercube to the random network. Suppose the nodes are distributed uniformly in the name space, and every node in the network generates one query for file 0. Suppose the requested file is cached according to the LCP-replication scheme at the nodes whose IDs are less than 128. The number of such nodes is a random variable, which is a function of the network instance.

Figure 2 (a) shows the query count at each node, averaged over 1000 instances of the random network. The horizontal axis shows the nodes in increasing order of their IDs. We see that the cache nodes each serve nearly the same number of queries, about 32 queries. As a comparison, without caching, the only server would serve 500 requests. Non-cache nodes also see the queries but do not serve them. As expected, the query count decreases roughly by half between consecutive groups of nodes. In summary, LCP-replication achieves perfect load balancing in the average sense, where the average is taken over instances of the network.

Figure 2 (b) shows the query count at each node in an instance of the random network. The load to any cache server is significantly less than 500 queries. However, the number

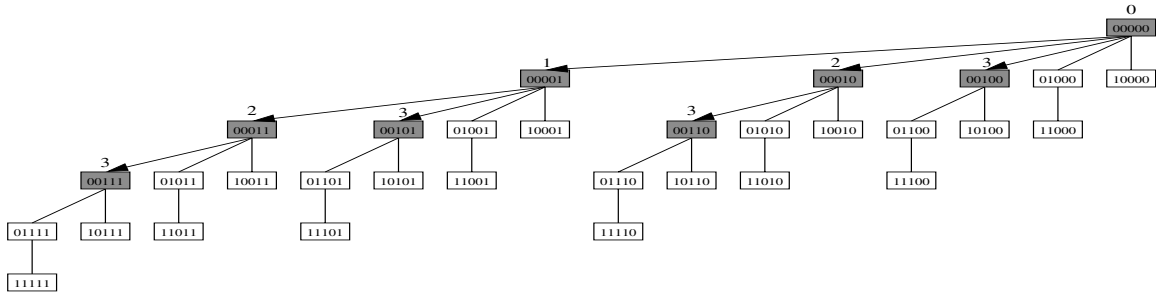


Fig. 1. An example of LCP-replication. The shaded nodes are cache nodes.

of queries served by different cache servers varies between 4 to 67. The *maximum load over all cache servers* is 67, about twice as much as the *average load across all cache servers*.

The average load-balancing performance is an inadequate metric for our case, because we do not repeatedly draw different network instances. Once a network instance is generated, it will be put to use. Except the gradual node arrivals and departures, the network remains more or less fixed for a long time. In the actual network instance to be used, the discrepancy between the maximum load and the average load is a cause of concern. We can show that, for larger but practical system sizes, the maximum load can be up to 10 times the average load, but is highly unlikely to be more than that. The methodology for such result on larger networks is described in Section IV. Our next task is to reduce this discrepancy to be less than 3 in virtually all network instances using a multiple-choice algorithm, which will be explained.

### B. Multiple-Choice Randomized Routing Tables

In this section, we consider the *multiple-choice random network construction algorithm*, an improvement to the single-choice algorithm. The motivation is to reduce the statistical fluctuation in the load to the cache nodes. We defer the discussion about its rationale and about how to evaluate its performance in large networks to Section IV. One related work that also uses the multiple-choice strategy for random network construction is [20]. The goal there is to make the distribution of the nodes in the name space more uniform.

Each node of the network keeps  $e$  counters. Consider an arbitrary node, denoted by  $v$ , whose ID is  $b_{e-1}...b_0$ . The counters are denoted by  $v[1], \dots, v[e]$ . Each  $v[i]$  counts the number of upstream nodes of the form  $b_{e-1}...b_{e-i+1} * ... *$ , i.e., the nodes that share at least  $(i-1)$ -digit prefix with  $v$ , that make  $v$  a next-hop node at level  $i$  in their routing tables.

For an arbitrary node,  $a_{e-1}...a_0$ , let us consider how to fill its routing table that conforms to the generic routing table shown in Table I. Recall that each next-hop entry in Table I specifies the format of the so-called *eligible nodes* for the corresponding level and the corresponding digit value. We will focus on the next-hop entry for level- $i$  and digit value  $c$ , where  $1 \leq i \leq e$  and  $c = 0$  or  $1$ . If  $c$  is equal to the  $i^{\text{th}}$  digit of the current node's ID, i.e.,  $c = a_{e-i}$ , we leave that entry unfilled since it won't be used by the routing algorithm.

Let us denote the complement of the binary digit  $a_{e-i}$  by  $\bar{a}_{e-i}$ . If  $c = \bar{a}_{e-i}$ , we choose  $d$  nodes, where  $d \geq 1$ , uniformly at random from all eligible nodes for the entry, that is, nodes of the form  $a_{e-1}...a_{e-i+1}\bar{a}_{e-i} * ... *$ . The  $d$  choices

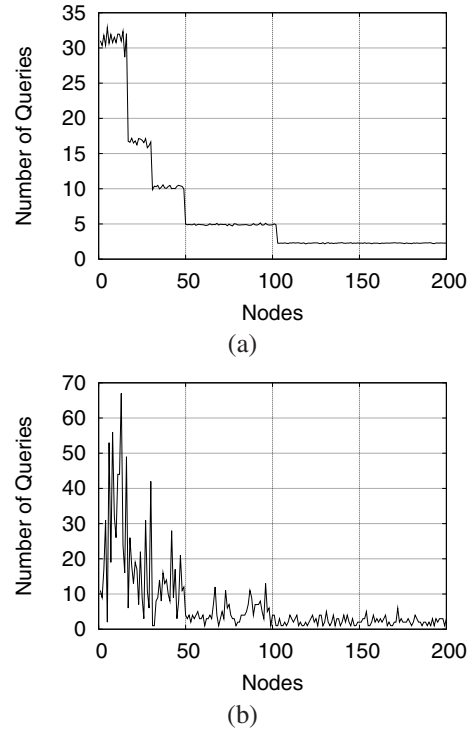


Fig. 2. Number of queries seen by each node in the random network.  $m = 5000$ ,  $n = 500$ . With LCP-replication. The file is at all nodes whose IDs are less than 128. (a) Averaged over 1000 network instances; (b) in an instance of the random network

are independent from each other and with replacement, i.e., there can be repetition among the choices. Let us denote the  $d$  choices by  $v_1, \dots, v_d$ . We then select among the  $d$  nodes the one with the least  $i^{\text{th}}$ -counter value,  $v_j[i]$ , and make it the next-hop node in the routing table entry being considered. In other words, the node selected to fill the routing table entry, denoted by  $v_l$ , satisfies  $v_l[i] = \min_{1 \leq j \leq d} v_j[i]$ . Finally, we increment  $v_l[i]$  by 1. The above procedure is summarized in Algorithm 3. Each node can run it independently and distributedly. The notation assumes that the current node ID is  $a_{e-1}...a_0$  and that  $T[i][c]$  is the next-hop node in the routing table entry corresponding to level  $i$  and digit value  $c$ . The statement  $T[i][c] \leftarrow v_l$  sets  $v_l$  as the next-hop node for this routing table entry.

Figure 3 shows the query counts at the nodes in a network

**Algorithm 3** Multiple-Choice Random Network Construction

---

```

for each level  $i$  from 1 to  $e$  do
  for each value  $c \in \{0, 1\}$  do
    if  $c \neq a_{e-i}$  then
      choose  $d$  eligible nodes independently, uniformly at
      random and with replacement, denoted by  $v_1, \dots, v_d$ 
       $l \leftarrow \operatorname{argmin}_{1 \leq j \leq d} v_j[i]$ 
       $T[i][c] \leftarrow v_l$ 
       $v_l[i] \leftarrow v_l[i] + 1$ 
    end if
  end for
end for

```

---

instance. The parameters are  $m = n = 5000$ ,  $k = 32$ , where  $k$  is the number of cache servers. The file is contained at cache nodes 0 to 31. The number of multiple choices is  $d = 1, 5, 10$  and 50 for different curves. The case of  $d = 1$  corresponds to the single-choice network. The average load over all cache nodes is equal to  $n/k = 156.25$  for all network instances, since we assume every node generates one query for file 0. The maximum load for  $d = 1$  is about 390. It can be reduced as  $d$  increases. At  $d = 5$  or 10, the load to the cache nodes is fairly well balanced. At  $d = 50$ , the load to the cache nodes, as well as all other nodes, starts to approach the average behavior (averaged over network instances).

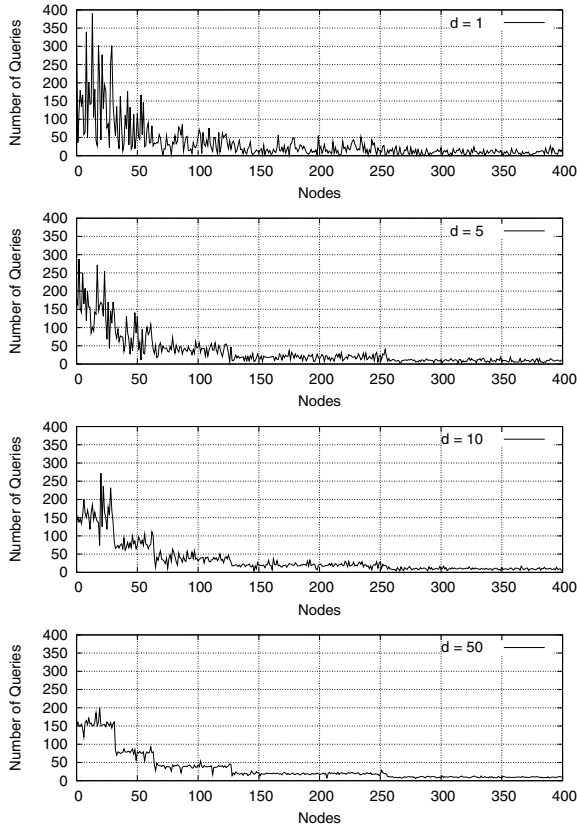


Fig. 3. Query counts at the nodes in a network instance.  $m = 5000$ ,  $n = 5000$ . With LCP-replication,  $k = 32$ . The file is contained at node 0 to node 31.

The remaining important concern is: How does the above

results change as the random network becomes large? This will be the subject of Section IV.

#### IV. MODEL OF THE RANDOM NETWORK: RECURSIVE BALLS IN BINS (RBNB)

When queries are generated uniformly by the nodes, Theorem 4 gives no information about the variation of the load across the cache nodes and across different network instances. Of particular interest is the load of the most loaded cache node, the maximum load. In this section, we will recast the random network into a balls-in-bins (BNB) model for easy understanding of such issues.

One benefit of making such a connection is that we can simulate much larger BNB problem due to its simplicity. In addition, the known theoretical results [25], [26], [27], [28] about the BNB problem allow us to interpret the simulation results with confidence and to extrapolate them for even large network size. For instance, we will eventually make performance-related conclusions about networks with  $2^{64}$  nodes. More importantly, the multiple-choice BNB problem has motivated the invention of the multiple-choice random network construction algorithm, which drastically improves the load-balancing performance for any fixed instance of the random network.

##### A. Equivalence of the RBNB Problem and the Single-Choice Random Network Construction Problem

To simplify the presentation, let us assume the name space size,  $m$ , is the same as the number of nodes,  $n$ , and that  $m = n = 2^e$  for some natural number  $e$ . Also assume every node generates exactly one query for file 0. With  $k$  cache nodes, the average load is always a constant,  $n/k$ . Theorem 4 says that the *expected number of queries* received by each cache node is  $n/k$ , where the expectation is taken over all network instances.

The problem of balls-in-bins (BNB) is to place  $n$  balls into  $k$  bins by selecting one destination bin uniformly at random and with replacement for each ball, and independently across different balls. Suppose  $k$  is also a positive integer of some power of 2. The *recursive balls-in-bins* (RBNB) problem is as follows. Initially, there are  $n$  bins, numbered  $0, 1, \dots, n-1$ , each with exactly one ball. In the first step, for each ball in bin  $i$ , where  $n/2 \leq i \leq n-1$ , a destination bin is chosen for it from bins  $0$  through  $n/2-1$ , uniformly at random and with replacement. The ball in bin  $i$  is placed into the chosen destination bin. This is repeated independently for each  $i$ ,  $n/2 \leq i \leq n-1$ . At the end of the first step, all balls are in bins from  $0$  to  $n/2-1$ . At the beginning of the general  $j^{\text{th}}$  step, for  $1 \leq j \leq \log_2(n/k)$ , all balls are in bins from  $0$  to  $n/2^{j-1}-1$ . For each bin  $i$  in the second half, i.e.,  $n/2^j \leq i \leq n/2^{j-1}-1$ , we randomly choose a destination bin from the first half, i.e., from  $0$  through  $n/2^j-1$ , and place all balls in the former bin into the destination bin. The process repeats  $\log_2(n/k)$  steps and all balls end up in bins  $0$  through  $k-1$  (See Figure 4).

The RBNB problem is the same as our single-choice random network construction problem. To see this, consider the subnetwork that consists of all the nodes and edges on the routes to node 0, which is an embedded tree rooted at node 0, with all edges directed toward the root. The construction of this subnetwork can be described as follows. In the first step, every node of the form  $1a_{e-2}\dots a_0$  selects a node of

the form  $0a_{e-2}\dots a_0$  randomly as its next-hop neighbor. In the  $j^{\text{th}}$  step, every node of the form  $0\dots 01a_{e-1-j}\dots a_0$  selects a node of the form  $0\dots 00a_{e-1-j}\dots a_0$  randomly as its next-hop neighbor. The process repeats  $\log_2(n/k)$  times. Describing the network construction in synchronized, ordered steps is not essential but serves to identify the network construction with the RBNB problem. Any asynchronous and distributed network construction algorithm following the same neighbor-selection rule results in the same network.

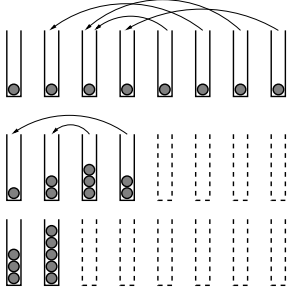


Fig. 4. Example of RBNB with two recursion steps.  $m = n = 8$ ,  $k = 2$ .

### B. Multiple-Choice RBNB

An advantage of recasting the random network construction problem into the RBNB problem is that, the BNB problem and its variants have been well studied. For instance, it is well known that, with  $k = n$ , the number of balls in the bin with the most balls, which will also be called 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 will be called the average load, is 1 in this case. One of the variants of the BNB problem is the multiple-choice BNB problem. In such a problem, for each ball,  $d$  bins are independently and randomly selected, with replacement, and their contents are examined. The 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. With multiple choice, there is an exponential reduction in the maximum load.

We are not aware of similar theoretical results for the RBNB or the multiple-choice RBNB problems. The techniques for proving the aforementioned results in the BNB case do not seem to apply to the RBNB case. However, the apparent relatedness of the RBNB and the BNB problems leads us to conjecture that, in RBNB, the ratio between the maximum load and the average load is not large, and that, multiple-choice RBNB can reduce the ratio significantly. We will substantiate the conjectures with simulation experiments.

We may distinguish two versions of the multiple-choice RBNB algorithm. Recall that, in a general step of the RBNB algorithm, some of the bins are source bins and some are destination bins and the objective is to move the balls in each source bin into a destination bin. In the *ball-count* version, each destination bin keeps track of the number of balls it currently contains. The balls in a source bin are assigned to the destination bin with the least number of balls among  $d$  choices. In the *bin-count* version, each destination bin keeps track of the number of *source bins* it has already consolidated. The balls in a source bin are assigned to the destination bin

with the least number of consolidated bins among  $d$  choices. The ball-count RBNB algorithm apparently should be more effective in reducing the maximum load than the bin-count algorithm. However, the bin-count algorithm is more easily applicable to the random network construction problem. Our algorithm in Section III-B (Algorithm 3) is equivalent to bin-count RBNB. The performance difference of the two versions is observable but not significant.

In the more general setting where  $m > n$ , the initial number of bins is  $m$ . Only  $n$  of them contains one ball each and the rest are empty. The RBNB algorithm needs a minor and straightforward modification. For brevity, we omit the details.

### C. Evaluation of Single-Choice and Multiple-Choice RBNB

We have conducted extensive simulation-based study characterizing the maximum load in the RBNB problem. For brevity, we will only highlight a few key results.

1) *The maximum load in a single run of RBNB*: Figure 5 (a) shows the maximum load vs. the average load,  $n/k$ , for the bin-count RBNB algorithm on log-log scale. Here,  $m = n = 2^{23}$  and  $k$  takes all possible values from  $2^{22}$ ,  $2^{21}$ , ...,  $2^0$ . The different curves correspond to  $d = 1, 2, 3$  and  $10$ , where  $d$  is the number of choices. For comparison purpose, we also plot the result for the BNB algorithm, and the straight line with slope 1 corresponding to the case that the maximum load is equal to the average load. The data for each curve is collected over a single run of the algorithm, and hence, each maximum load is an instance of a random variable.

The curves for the RBNB algorithm are not as smooth as the one for the BNB algorithm, indicating higher statistical fluctuation in the maximum load. But the fluctuation is small. For the plain RBNB algorithm ( $d = 1$ ), the maximum load is at most 7 times of the average load. Multiple choice is very effective in further reducing the difference. For instance, for  $k = 2^{16}$ , the maximum loads are 939 or 475 when  $d = 1$  or 2, respectively. The average load is 128. Overall, the maximum load is 1 to 4 times the average load when  $d = 2$ . When  $d = 10$ , the maximum load is no more than 2.5 times the average load.

Figure 5 (b) shows similar plots for the case where the name space is not fully populated, with  $m = 2^{23}$  and  $n = 2^{16}$ . For  $d = 1$ , the ratio of the maximum load to the average load is less than 10 for all experimented parameters. With a small number of multiple choices, the ratio substantially decreases. In particular, with  $d = 3$ , the ratios are all less than 3. An additional result not shown here is that, for fixed  $m$ , the ratio increases as  $n$  increases. The implication is that we can take the case of fully populated name space (i.e.,  $m = n$ ) as the worst-case scenario.

2) *Statistical fluctuation of the maximum load*: Previous RBNB experiments show only instances of the maximum load. What allows us to draw conclusions based on them is that the statistical fluctuation of the maximum load is not large. Multiple choice further reduces the fluctuation significantly. We now substantiate these claims.

We will focus on the cases where  $m = n = 2^{20}$  as an example. For each fixed  $k$  and  $d$ , we collect samples of the maximum load from 10000 different runs of the RBNB algorithm, and then derive the empirical probability mass function (pmf) from the histogram of the samples. In Figure 6, we compare the pmf's of the single-choice and multiple-choice

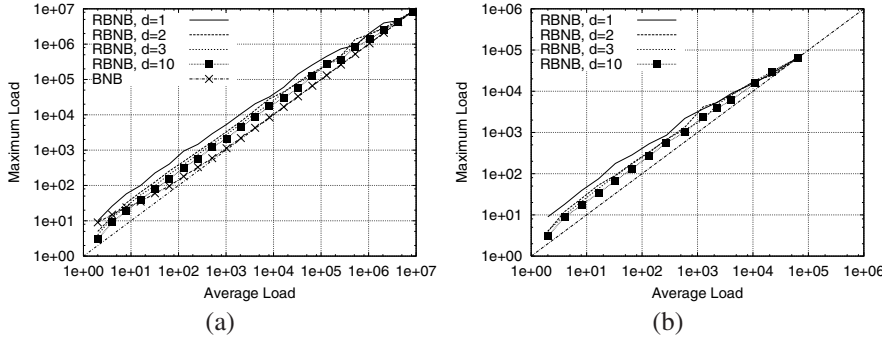


Fig. 5. Maximum load vs. average load for RBNB under the bin-count multiple-choice method. (a)  $m = n = 2^{23}$ ; (b)  $m = 2^{23}$ ,  $n = 2^{16}$ .

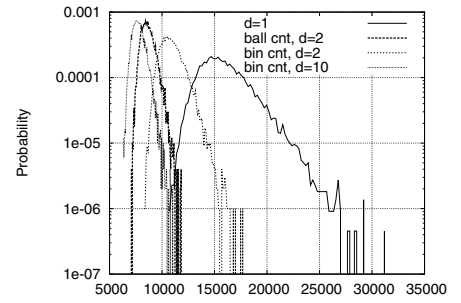


Fig. 6. Probability mass function of the maximum load for RBNB. Comparison of the single-choice and multiple-choice RBNB.  $m = n = 2^{20}$ ,  $k = 2^8$ .

RBNB. We see that, with or without multiple choice, the tail of the pmf decreases at least exponentially fast and the statistical fluctuation is not large compared to the respective mean of each case. For instance, the largest ratio observed between a sample of the maximum load and its mean is no more than 3, and this occurs for  $d = 1$ . With multiple choices, significant reduction occurs not only in the mean of the maximum load, but also in its statistical fluctuation. Furthermore, the tail of the pmf also decays faster.

An important question is how the statistical fluctuation of the maximum load varies with  $n$ . In order for the comparison to be fair, we must keep  $n/k$  constant. A result not shown here is that the statistical fluctuation decreases as  $n$  increases. This is not surprising, since it is known that, in the case of BNB, the maximum load is more and more concentrated at  $(1 + o(1)) \ln n / \ln \ln n$  as  $n$  approaches infinity.

3) *Trend of the expected maximum load versus  $n$* : We next discuss how the expected maximum load grows with  $n$ , the number of balls. We assume  $m = n$  throughout. The goal is to extrapolate the trend and make prediction about the size of the maximum load for cases with much larger  $n$ , too large to be observed directly by simulation.

Figure 7 (a) shows how the expected maximum load grows with  $\log_2 n$  for the bin-count RBNB algorithm. To be able to compare the results for different  $n$ , we keep  $n/k$  constant at  $2^6$ , so that the resulting average load across the bins at the end of the algorithm is kept at 64. The reason we choose such a value  $2^6$  is that, according to another study not shown here, the ratio of the expected maximum load to the average load is the largest when  $k$  is close to  $n$ . We observe that, for each  $d$ , the growth of the expected maximum load is slower than a linear function of  $\log_2 n$ . Using multiple choices ( $d > 1$ ) in the RBNB algorithm can drastically reduce the maximum load. This is true for even small  $d$ , e.g.,  $d = 2$  to 5.

The real significance of multiple choice reveals itself when we plot the same set of results with the  $x$ -axis in log scale, as in Figure 7 (b). A straight line in this plot corresponds to a linear function in  $\log \log n$ . We see that, for  $d = 1$ , the growth of the expected maximum load is faster than  $\log \log n$ , while for  $d > 2$ , the growth is no faster than  $\log \log n$ . This phenomenon is believable in view of the similarity between the multiple-choice RBNB problem and the multiple-choice BNB problem.

With Figure 7 (a) and (b), if the trend of the expected

maximum load continues, we can find its upper bound for larger values of  $n$  by fitting a linear function that grows no slower than the respective curve. We have done this for the case of  $n = 2^{64}$ , a large enough number for most practical systems, but too large to be observed by direct simulation. The result is presented in Figure 8, where we normalize the expected maximum load against the average load. The expected maximum load is bounded from above by 1166, 582, 270, 238, 219, and 184 for  $d = 1, 2, 5, 10, 20$  and 50, respectively. The corresponding ratios of the expected maximum load to the average load are 18.2, 9.1, 4.2, 3.7, 3.42 and 2.9. For  $n$  up to  $2^{64}$ , a small value of  $d$ , e.g.,  $d \leq 5$ , gives us most of the reduction in the expected maximum load. In addition, the ratio is not sensitive to  $n/k$ , indicating that similar results hold for a wider range of values for  $k$ .

## V. CONCLUSIONS

This paper presents several Plaxton-type networks that have excellent load-balancing features, in addition to the usual properties of allowing prefix routing and having small network diameters. Our most original and important contribution is the invention of the multiple-choice random network. It was first motivated by our study of load balancing on the hypercube network. We observed that LCP-replication on the hypercube leads to perfectly balanced load across the cache servers. For the same number of requests, this requires the smallest number of cache servers, the smallest number of replicas, the smallest amount of replication traffic, and the replication process takes the shortest time. We also observed that there is a simple, distributed, on-demand caching algorithm, LCP-caching, that achieves the same performance results as LCP-replication.

However, the hypercube can be restrictive for practical purpose. It requires the size of the name space to be a power of 2, and also fully populated by nodes. We discover that a far more practical and flexible Plaxton-type network with randomized routing tables can accommodate the same LCP-replication and caching algorithms and enjoys similar load-balancing performance, except that the load is perfectly balanced after averaged over many network instances. For fixed network instances, the maximum load over all cache servers can be up to 10 times the average load.

The multiple-choice random network drastically reduces this ratio in virtually any network instance. Its invention is inspired by our discovery of its connection with the recursive balls-



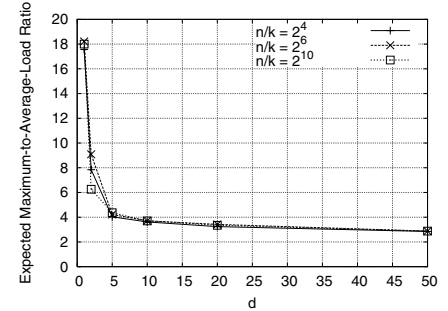
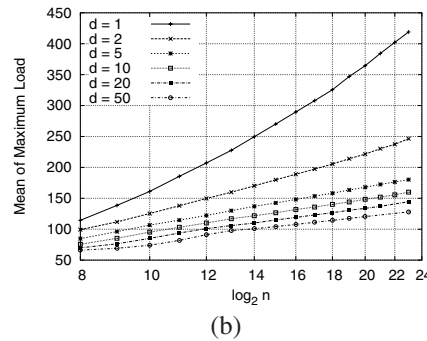
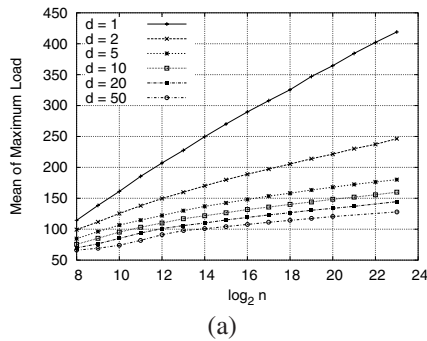


Fig. 7. Expected maximum load versus  $\log_2 n$  for bin-count RBNB. We keep  $n/k = 2^6$  for all cases. The average load across the bins is 64. (a)  $x$ -axis in linear scale; (b)  $x$ -axis in log scale.

Fig. 8. Ratio of the expected maximum load to the average load for bin-count RBNB.  $n = 2^{64}$ .

in-bins problem. The connection also allows us to conduct simulation experiments on large networks, to interpret the simulation results with confidence, and to extrapolate them for even larger network sizes, e.g., with  $2^{64}$  nodes.

Due to the complexity of any load-balancing system, many system-level details that we have considered are not discussed in this paper. These include, for instance, the joining-and-departure dynamics of the nodes, the case of multiple files, and the case of heterogeneous server capacities. We expect no fundamental difference between our basic design and other published designs with respect to the degree of difficulty in incorporating these details.

#### REFERENCES

- [1] D. Li, X. Lu, and J. Wu, "FISSIONE: A scalable constant degree and low congestion DHT scheme based on Kautz graphs," in *Proceedings of IEEE Infocom 2005*, Miami, FL, March 2005.
- [2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proc. ACM SIGCOMM '2001*, San Diego, CA, August 2001, pp. 149–160.
- [3] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, Jan. 2004.
- [4] A. Rowstron and P. 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.
- [5] S. Ratnasamy, P. Francis, M. Hanley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM '2001*, San Diego, CA, August 2001, pp. 161–172.
- [6] F. Kaashoek and 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, Feb. 2003.
- [7] D. Loguinov, A. Kumar, V. Rai, and S. Ganesh, "Graph-theoretic analysis of structured peer-to-peer systems: Routing distances and fault resilience," in *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.
- [8] D. Malkhi, M. Naor, and D. Ratajczak, "Viceroy: A scalable and dynamic lookup network," in *ACM Symposium on Principles of Distributed Computing (PODC 2002)*, Monterey, CA, July 2002.
- [9] A. Kumar, S. Merugu, J. Xu, and X. Yu, "Ulysses: A robust, low-diameter, low-latency peer-to-peer network," in *Proceedings of the 11th IEEE International Conference on Network Protocols (ICNP'03)*, Atlanta, Georgia, USA, Nov. 2003.
- [10] J. Xu, A. Kumar, and X. Yu, "On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 151–163, Jan. 2004.
- [11] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. 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.
- [12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. 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.
- [13] A. Rowstron and P. 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, Oct 2001, pp. 188–201.
- [14] 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)*, Newport, Rhode Island, June 1997, pp. 311–320.
- [15] Y. Xia, S. Chen, and V. Korgaonkar, "Load balancing with multiple hash functions in peer-to-peer networks," in *Proceedings of The Twelfth International Conference on Parallel and Distributed Systems (ICPADS 2006)*, Minneapolis, USA, July 2006.
- [16] M. Adler, E. Halperin, R. M. Karp, and V. V. Vazirani, "A stochastic process on the hypercube with applications to peer-to-peer networks," in *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, San Diego, CA, June 2003, pp. 575–584.
- [17] D. R. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in *Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2004)*, Barcelona, Spain, June 2004.
- [18] J. W. Byers, J. Considine, and M. Mitzenmacher, "Geometric generalizations of the power of two choices," in *Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2004)*, Barcelona, Spain, June 2004.
- [19] X. Wang, Y. Zhang, X. Li, and D. Loguinov, "On zone-balancing of peer-to-peer networks: Analysis of random node join," in *Joint International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS 2004)*, New York, June 2004.
- [20] K. Kenthapadi and G. S. Manku, "Decentralized algorithms using both local and random probes for P2P load balancing," in *Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, Las Vegas, Nevada, July 2005.
- [21] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured P2P systems," in *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, Feb. 2003, pp. 311–320.
- [22] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in *Proceedings of IEEE Infocom 2004*, Hong Kong, March 2004.
- [23] V. Ramasubramanian and E. G. Sifer, "Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays," in *Symposium on Networked Systems Design and Implementation (NSDI 2004)*, San Francisco, USA, March 2004.
- [24] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.
- [25] N. L. Johnson and S. Kotz, *Urn Models and Their Application*. John Wiley & Sons, 1977.
- [26] G. H. Gonnet, "Expected length of the longest probe sequence in hash code searching," *Journal of the ACM*, vol. 28, no. 2, pp. 289–304, April 1981.
- [27] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, "Balanced Allocations," *SIAM Journal on Computing*, vol. 29, no. 1, pp. 180–200, February 2000.
- [28] M. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing," Ph.D. dissertation, University of California, Berkeley, 1996.