A Stochastic Process on the Hypercube with Applications to Peer-to-Peer Networks

[Extended Abstract]

Micah Adler*
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003-4610,
USA
micah@cs.umass.edu

Eran Halperin†
International Computer Science Institute
and Computer Science Division
University of California
Berkeley, CA 94720.
eran@cs.berkeley.edu

Richard M. Karp
International Computer Science Institute
Berkeley, CA 94704, USA,
and Computer Science Division
University of California
Berkeley, CA 94720,USA
karp@cs.berkeley.edu

Vijay V. Vazirani
College of Computing,
Georgia Institute of Technology,
Atlanta, GA 30332-0820.
vazirani@cc.gatech.edu.

ABSTRACT

Consider the following stochastic process executed on a graph $G = (V, E)$ whose nodes are initially uncovered. In each step, pick a node at random and if it is uncovered, cover it. Otherwise, if it has an uncovered neighbor, cover a random uncovered neighbor. Else, do nothing. This can be viewed as a structured coupon collector process. We show that for a large family of graphs, $O(n)$ steps suffice to cover all nodes of the graph with high probability, where n is the number of vertices. Among these graphs are d-regular graphs with $d = \Omega(\log n \log\log n)$, random d-regular graphs with $d = \Omega(\log n)$ and the k-dimensional hypercube where $n = 2^k$.

This process arises naturally in answering a question on load balancing in peer-to-peer networks. We consider a distributed hash table in which keys are partitioned across a set of processors, and we assume that the number of processors grows dynamically, starting with a single processor. If at some stage there are n processors, the number of queries required to find a key is $\log_2 n + O(1)$, the number of pointers maintained by each processor is $\log_2 n + O(1)$, and moreover the worst ratio between the loads of processors is $O(1)$, with high probability. To the best of our knowledge, this is the first analysis of a distributed hash table that achieves asymptotically optimal load balance, while still requiring only $O(\log n)$ pointers per processor and $O(\log n)$ queries for locating a key; previous methods required $\Omega(\log^2 n)$ pointers per processor and $\Omega(\log n)$ queries for locating a key.

Categories and Subject Descriptors

F.3 [**Mathematics and Computing**]: Probability and Statistics—*Stochastic processes*; D.1 [**Data**]: Data Structures—*Distributed data structures*

General Terms

Algorithms

Keywords

Peer to peer, coupon collector, hypercube, hash table, load balancing

1. INTRODUCTION

In the past few years, there has been a considerable amount of research on peer-to-peer applications. A central problem for such applications is developing a distributed protocol that finds a data item stored on one of a potentially large number of processors. We can think of this as a distributed hash table: given a key, we want to hash that key to a value

*Supported by NSF Research Infrastructure Award EIA-0080119 and by NSF Faculty Early Career Development Award CCR-0133664

†Supported in part by NSF grants CCR-9820951 and CCR-0121555 and DARPA cooperative agreement F30602-00-2-0601.

STOC'03, June 9–11, 2003, San Diego, California, USA.

that identifies the processor that holds that key. We also require an efficient method for finding that processor. This hash table should function without centralized control and should also provide an efficient technique for dealing with the frequent and unpredictable arrival and departure of the processors that store the table. Research that has addressed this problem includes [13, 11, 12, 10, 4, 6, 8].

In this paper, we consider distributed hash tables of the following type. Each key is hashed to a real number in the interval $[0, 1]$. The task of storing this interval is partitioned across the processors. In particular, the processors are organized using a (not necessarily complete) binary tree, where each leaf of the tree corresponds to a processor. Each leaf is also assigned a binary string describing the path from the root to that leaf, with left branches represented by a 0 and right branches represented by a 1. Let $S(i)$ be the string assigned to the ith leaf in the left to right ordering of leaves. Let n be the number of processors, and let $N(i)$ be the real number (in $[0, 1)$) obtained by interpreting the string $S(i)$ as a binary decimal. The processor corresponding to leaf $i < n$ is responsible for storing items that hash to the real interval $[N(i), N(i+1)]$. In addition, the processor corresponding to leaf 1 is also responsible for the real interval $[0, N(1)]$. The processor corresponding to leaf n is responsible for the real interval $[N(n), 1]$.

In order to allow navigation through this data structure, processors store pointers to each other. In particular, the processor at leaf j maintains a pointer to the processor at leaf i if $S(i)$ is a prefix of a string obtained from $S(j)$ by flipping one bit in $S(j)$ and appending it with zeros. In Figure 1 we show an example of such a tree, illustrating the pointer set of one of the vertices. Such pointers can be thought of as a generalization of the hypercube. Using these pointers, if a search for an item starts at any processor, the hashed item can be found using at most h queries to processors, where h is the height of the tree (our analysis shows that $h = \log_2 n + O(1)$). Also note that the number of pointers needed for each processor is at most h. These pointers are also used to handle processor arrivals and departures. An arrival is accomplished by first finding an appropriately chosen leaf of the tree, and then performing a *split* operation on that leaf. A split of leaf i replaces i with an internal node, as well as two children of that node. The processor that previously resided at leaf i is assigned to one child, and the new arrival is assigned to the other child. Storage of the content hashed to the original region as well as the pointers held by processors are adjusted appropriately. The process for choosing which leaf to split is central to this paper, and will be discussed below. We further suggest a departure strategy which seems to work well in practice, although it is left open to obtain a rigorous analysis for that process.

We shall refer to this distributed hash table as the *hypercubic hash table*. The hypercubic hash table is motivated by the hash table introduced in [11]. Instead of the hypercubic structure we consider, [11] considers a d-dimensional torus, where each processor is responsible for a subregion of the torus, and maintains pointers to processors that are responsible for neighboring regions. Arrivals are handled by splitting an existing region in half along one dimension. The hypercubic hash table we consider here can be viewed as the structure from [11], where the dimension d is chosen to be so large that no dimension is ever split more than once.

There are a number of measures of the performance of a distributed hash table. Of primary interest are the number of queries required to locate a requested item, the number of pointers each processor must store, as well as how well the storage load is balanced between the processors. The distributed hash tables of [4], [12], [10], and [13] all require $O(\log n)$ queries for location and $O(\log n)$ pointers per processor for a data structure with n processors. The hash table of [11] requires $O(dn^{1/d})$ queries for location, and $O(d)$ pointers per processor. The hash table of [8] requires $O(\log n)$ queries, $O(1)$ pointers in expectation, and $O(\log n)$ pointers per processor with high probability.

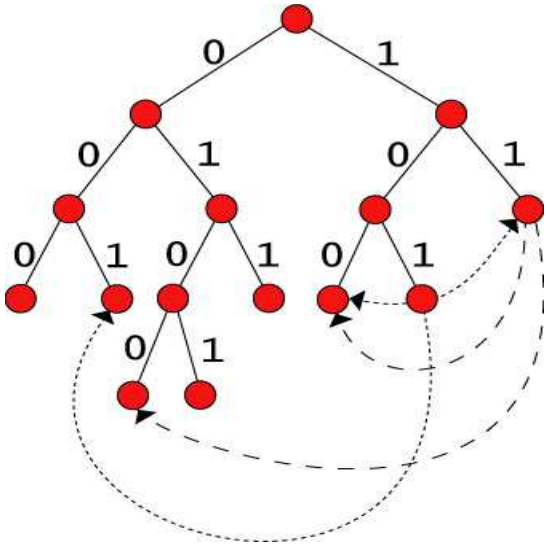

Figure 1: An example of a tree containing the strings $000, 001, 0100, 0101, 011, 100, 101, 11$. Each of the hosts is located in one of the leaves. The tree is virtual in the sense that its edges do not exist in the network. The dotted lines are the actual connections in the network. In this case, the host corresponding to the leaf 11 has pointers to the hosts located in 100 and 0100. These neighbors are determined by flipping one of the bits in the string 11, and padding it with zeros at the end.

In this paper, we concentrate on the third indication of performance: load balancing. The analysis of this measure is typically more involved than that of the other two described above, and it has seen much less attention in the existing work. Let $V_{max}(D)$ be the maximum fraction of the range of the hash function stored on any processor with hash table D, and let $V_{min}(D)$ be the corresponding minimum fraction. We say that a distributed hash table D achieves load balance α if $V_{max}(D)/V_{min}(D) = \alpha$. For example, the hypercubic hash table we consider achieves load balance 1 if the underlying binary tree is complete. Note that the fraction of the hash function range is an appropriate measure of load balance, since we expect the number of hashed items to be much larger than the number of processors.

The actual load balance achieved by the hypercubic hash table will depend on the rule for choosing which node of the tree to split on a processor arrival. The "vanilla" version of the [11] scheme suggests a strategy for choosing a region to split that, in our framework, is equivalent to choosing a

value in the interval $[0,1]$ uniformly at random, and then splitting the leaf of the binary tree that is responsible for the region containing that value. Such a rule can be easily implemented in a distributed fashion, but it is not difficult to show that the resulting distributed hash table on n nodes achieves load balance $\Omega(\log n)$. The distributed hash tables of [4], [12], and [10] use similar splitting rules. The scheme from [13] also achieves load balance $\Omega(\log n)$. They describe how to improve this to load balance $O(1)$, but this requires the number of pointers stored at each processor to be increased to $\Theta(\log^2 n)$.

In this paper, we study a second technique suggested in [11] for choosing which node to split. A leaf node is chosen randomly as before. However, instead of splitting that leaf node, we consider the set consisting of that node as well as all the leaves that the processor assigned to the chosen leaf has pointers to. The leaf node in this set with minimum distance from the root is split. If there is more than one such node, the originally chosen node is given preference, and, if that node does not have the minimum distance, one of the nodes that do is chosen uniformly at random. This splitting rule is shown in [11] to perform quite well through simulations, but no analytical evidence for this has existed prior to the work described here. We demonstrate that the resulting hypercubic hash table achieves constant load balance. In particular, we demonstrate that there are constants c_1 and c_2 such that after n split operations have been performed, with high probability the maximum height of any node in the tree is $\log_2 n + c_1$, and the minimum height is $\log_2 n - c_2$.

To the best of our knowledge, this is the first analysis of a distributed hash table that achieves asymptotically optimal load balance, while still only requiring $O(\log n)$ queries for location, and $O(\log n)$ pointers per processor.[1] Furthermore, the rule we consider for performing a split does not affect the number of queries required for the split by more than a constant factor. In particular, the number of queries required to find a randomly chosen initial node is $O(\log n)$; the improved rule for choosing which node to split only requires querying the $O(\log n)$ neighbors of this node.

In order to prove the above bounds, we introduce the following random process on the k-dimensional hypercube. In this process, each node could be either covered or uncovered. The initial state is when all nodes are uncovered. At each step of the process we pick a random node v of the hypercube. If v is uncovered, we cover it. If v is covered and all its neighbors are also covered, we do nothing. Otherwise, we randomly pick one of its uncovered neighbors and cover it. We call this process Process 1. We show that with high probability, Process 1 covers all vertices of the hypercube after $O(n)$ steps, and each of the first $\Omega(n)$ steps covers an uncovered node, where $n = 2^k$. We then use these bounds to show that the tree representing the hypercubic hash table is balanced, which provides the bounds claimed above.

Process 1 can be defined on any graph $G = (V, E)$. If the graph is an independent set, this is exactly the coupon collector process and therefore, with high probability, the process covers all nodes after $O(n \log n)$ steps. We show that our proof techniques can be extended to other graphs: we prove that for every d-regular graph, Process 1 covers all vertices after $O(n + n\frac{\log n \log d}{d})$ steps. We also show that for random d-regular graphs the stronger bound of $O(n + n\frac{\log n}{d})$ holds. These extensions may be useful in the design of different algorithms and data structures. We note that independently, Alon [1] proved that for any (n, d, λ)-graph[2] G, $O(n+n(\frac{\lambda}{d})^2 \log n)$ steps are sufficient. Alon further shows that for any d-regular graph on n vertices, the expected number of steps that Process 1 uses until it covers the whole graph is at least $n - \frac{n}{d} + \frac{n}{d}\log_e(n/d)$. Alon's proof also gives a bound of $n+(1+o(1))n \log_e n/d$ steps for random d regular graphs.

This paper is organized as follows: in the remainder of this section, we provide more details concerning other proposed distributed hash tables, as well as other related work. In Section 2, we reduce the task of analyzing the performance of the hypercubic hash table to the problem of analyzing a simple stochastic process on the hypercube. In Section 3, we analyze this hypercubic process. In Section 4, we describe extensions of our techniques to other graphs. In Section 5 we provide an alternative proof of the bound on the cover time of the random process on the hypercube. In Section 6 we describe the results of some simulations.

1.1 Previous Work

In the distributed hash table of [10], each participating processor i is assigned a random k-bit string $S(i)$, where k is large enough that it is unlikely for two processors to have the same string. Each hashed key is stored on the processor i such that $S(i)$ is closest to its hash value in lexicographic ordering. Note that choosing a random k-bit string for an arrival is similar to the version of the hypercubic hash table where a new arrival is handled by splitting a randomly chosen node (without regard to the neighbors of that node). The main difference is that the entire string $S(i)$ stays with the processor in the technique of [10]; with the technique we consider here only the portion of the randomly chosen real number that describes the first currently unsplit node of the binary tree is ever used.

The rule used in [10] for choosing which pointers between processors to maintain can also be thought of as a generalization of the hypercube. This rule takes into account locality in the underlying network, and thus has the advantage that the real network length of a path in the data structure will typically be considerably shorter than if this measure is not taken into account. The distributed hash table of [10] is static (i.e., it does not allow for the arrival and departure of processors), but techniques for making it dynamic have been described in [12], [4], and [6].

In the distributed hash table of [13], data items and processors are hashed to integers chosen uniformly at random from the range $[0, 2^k - 1]$. Each data item is stored on the processor with the next largest hash value (mod 2^k) to the data item's hashed value. In order to locate data items, each processor stores $\log n$ pointers: for each integer j, $0 \leq j \leq \log n - 1$, the processor with the ith largest hash value stores a pointer to the $(i + 2^j \mod 2^k)$th largest hash value. It is not difficult to show that this data structure achieves load balance $\Theta(\log n)$ with high probability. This can be improved to $O(1)$ by simulating $\log n$ virtual processors on each actual processor. However, this increases the

[1] We point out that while [8] does not address the issue of load balance at all, their distributed hash table could also be used to achieve a similar result by using a technique similar to that used to improve the load balance of [13].

[2] G is an (n, d, λ)-graph if G is a d regular graph on n vertices, such that the absolute value of every eigenvalue of the adjacency matrix of G, besides the largest one, is at most λ.

number of pointers that each real processor must store to $\Theta(\log^2 n)$.

The idea of using the best out of multiple possibilities to improve load balance has been studied extensively in the analysis of "balls into bins" games (see [7] and [3] for early work, and [9] for a recent survey). The scenario considered there, as well as the notion of load balance, is significantly different from this paper. However, the observation that more choices is helpful is similar.

2. THE STOCHASTIC PROCESS

In order to analyze the hypercubic hash table described in the previous section, we consider Process 1 which was defined in the introduction. Let $n = 2^k$ be the number of nodes in the hypercube. We show the following:

THEOREM 2.1. *Process 1 covers all the nodes of the hypercube in $O(n)$ steps with high probability; the probability of error is at most inverse polynomial in n.*

THEOREM 2.2. *There exists a constant c, such that with high probability, each of the first $\lfloor cn \rfloor$ steps of Process 1 cover an uncovered vertex; the probability of error is at most inverse polynomial in n.*

Observe that a bound of $O(n \log n)$ in Theorem 2.1 follows trivially from the coupon collector argument, since the above process dominates the coupon collector process. A bound of $O(n)$ steps is easy to show for the following modification of the coupon collector process: pick $\log n$ items at random and if any of them is not covered, cover a random uncovered item. In our case, the randomization is in the choice of the initial node – the $k = \log_2 n$ nodes that are examined subsequently are completely determined by this choice. On the other hand, each node has a "funnel" of size k, its k neighbors, that helps it get covered – we will use this critically in the proof.

The subsequent sections of this paper concern the analysis of this stochastic process. In the remainder of this section, we demonstrate the implications of Theorem 2.1 to the hypercubic hash table. We state these implications as bounds on the minimum and maximum height of any leaf in the underlying binary tree. From these bounds, it follows easily that after n split operations, the hypercubic hash table achieves asymptotically optimal load balance, while still only requiring $O(\log n)$ queries for location, and $O(\log n)$ pointers per processor.

COROLLARY 2.3. *There is a constant c_2 such that after n split operations have been performed on the hypercubic hash table, the probability that the minimum height of any node in the tree is less than $\log n - c_2$ is at most inverse polynomial in n.*

PROOF. At any point during the n split operations, consider some level ℓ of the tree such that every node at level $\ell - 1$ has been split, but there is some node at level ℓ that has not been split. It is easy to see that the process of inserting a new processor to the hash table is equivalent to the hypercube process in the following sense. The vertices of the hypercube are the nodes in level $\ell - 1$. A node is covered if it has already been split, and is uncovered otherwise. From Theorem 2.1, we know that there is a constant α such that after $\alpha 2^\ell$ more split operations have been performed, the probability that there are any unsplit nodes on level ℓ is inverse polynomial in 2^ℓ.

Let $c_2 = \log(4\alpha)$. For each level i of the tree, $1 \leq i \leq \log n - \log\log n - c_2$, we allocate $\alpha 2^i \log n$ split operations to level i. These splits are further subdivided into $\log n$ phases of $\alpha 2^i$ split operations each. Given that all nodes at level $i-1$ have been split before a given phase for level i, with probability at least $1/2$ there are no unsplit nodes at level i after that phase. Thus, if all nodes at level $i-1$ are split before the $\log n$ phases allocated to level i, the probability that i has any unsplit nodes after the split operations allocated to it is at most inverse polynomial in n.

For each level i of the tree, $\log n - \log\log n - c_2 < i \leq \log n - c_2$, we allocate $\alpha 2^i$ split operations to level i. This is sufficient that the probability that any such i has any unsplit nodes is also at most inverse polynomial in n. Note that we have allocated a total of at most n split operations. Since we are considering a total of only $\log n - c_2$ levels of the tree, the probability that any of the first $\log n - c_2$ levels of the tree fails to have every node split is still inverse polynomial in n. □

COROLLARY 2.4. *There is a constant c_1 such that after n split operations have been performed on the hypercubic hash table, the probability that the maximum height of any node in the underlying binary tree is larger than $\log n + c_1$ is at most inverse polynomial in n.*

PROOF. Consider the level $l = \log n + c_1$ for c_1 large enough. We can view the process of splitting vertices of level l as dominated by the hypercube process on the hypercube with 2^l vertices. By Theorem 2.2, the first $c2^l$ insertions do not split any vertex in level $l+1$ with high probability, where the probability for error is inverse polynomial in 2^l, that is, inverse polynomial in n. By setting $c_1 = \log_2(1/c)$, we get that with high probability we do not split any vertex in level $l+1$ in the first n insertions. Thus, the corollary follows. □

3. ANALYSIS OF THE STOCHASTIC PROCESS

In this section we prove Theorem 2.1.

3.1 An easier bound of $O(n \log\log n)$

A bound of $O(n \log\log n)$ is easier to establish and is quite instructive, so we will present it first.

We first need some notations and definitions. In all the statements below, "high probability" will mean one minus inverse polynomial in n probability; the exact polynomial depends on the multiplier of n chosen in the number of steps executed. An uncovered node will also be called *free*. For a vertex v, let $N_F(v)$ and $N_C(u)$ be the set of free and covered neighbors of v respectively. For a vertex v and a set A, let $d_v(A)$ be the number of neighbors of v in A. We will use the following Chernoff-like upper bound for large deviations (the proof can be found, e.g., in [2]).

LEMMA 3.1. *Let Z be a random variable with the binomial distribution, $Z \sim B(n,p)$. For every $a > 0$, prob$(Z < np - a) < e^{-a^2/(2np)}$.*

It is easy to see that $O(n)$ steps suffice to ensure that with high probability each node has at most $k/2$ uncovered

neighbors; in fact, this is true for the simpler coupon collector process as well. We will call this phase 0, and will execute $\log k$ more phases of $O(n)$ steps each, numbered $1, \ldots, \log k$. We say that phase i is successful if at the end of phase i, $|N_F(v)| \leq \frac{k}{2^{i+1}}$ for every vertex v. The claimed bound follows from:

LEMMA 3.2. *With high probability, for $i = 1, \ldots, \log k$, phase i is successful.*

PROOF. Assume that at the beginning of phase i, each node has at most $\frac{k}{2^i}$ uncovered neighbors. Consider a node v such that $|N_F(v)| \geq \frac{k}{2^{i+1}}$. We will show that at the end of phase i, with high probability, v has at most $\frac{k}{2^{i+1}}$ uncovered neighbors, hence proving the lemma.

Let L_1 denote the free neighbors of v, and L_2 the covered neighbors of L_1, though not including v.

We now consider a single step, and we show that as long as $|L_1| \geq \frac{k}{2^{i+1}}$, the probability to cover a vertex in L_1 is large. Clearly, in order to cover such a vertex, the process has to choose a vertex u of L_2, and then to choose one of its $d_u(L_1)$ neighbors in L_1. Recall that the number of free neighbors of u is bounded by $\frac{k}{2^i}$. We thus get that this probability is at least

$$\sum_{u \in L_2} \frac{1}{n} \cdot d_u(L_1) \frac{2^i}{k} = \frac{2^i}{nk} \sum_{v \in L_1} |N_C(v)| \geq \frac{2^{i-1}}{n} |L_1| \geq \frac{k}{4n}.$$

Let c be a large constant. Consider a sequence of cn steps of the algorithm, and let X be a random variable, where $X \sim B(cn, \frac{k}{4n})$. Clearly, as long as $|L_1| \geq \frac{k}{2^{i+1}}$, the number of vertices of L_1 covered by the process is dominating X, that is, the probability that after cn steps we still have $|L_1| \geq \frac{k}{2^{i+1}}$ is at most the probability that $X < \frac{k}{2^{i+1}}$. By Lemma 3.1 we get

$$\begin{aligned} prob(X < \frac{k}{2^{i+1}}) &\leq prob(X < \frac{ck}{4} - \frac{k}{4}(c - 1/2^{i-1})) \\ &\leq e^{-k(c-1/2^{i-1})^2/8c} < e^{-2k}, \end{aligned}$$

where the last inequality holds for c large enough. We now use the union bound, and since the total number of vertices is 2^k, and the total number of phases is $\log k$, the probability that phase i is not successful for some i is at most $\log k 2^k e^{-2k}$ which is inverse polynomial in n.

□

3.2 Exploiting differential progress via convexity

At any step during Process 1, different nodes will be at different stages of progress, where by *progress of a node* we mean the number of its covered neighbors. A shortcoming of the argument given above is that it has no way of taking advantage of nodes that are further along – it only utilizes the worst case progress made in previous phases to bound the number of steps needed in future phases.

From the proof of Lemma 3.2, it is easy to see that if each node of L_2 had $< k/2^{i+j}$ uncovered neighbors, then $O(n/2^j)$ steps would guarantee that with high probability the number of uncovered neighbors of v are halved. Lemma 3.3 below shows that this would be the case even if *on average* nodes in L_2 had fewer uncovered neighbors, thereby enabling us to argue about the progress made by aggregates of nodes.

Conduct a breadth first search starting with node v, and let $B_i(v)$ be the nodes at level i, $i = 1, \ldots, k$. W.l.o.g. assume that v is labeled with the k-bit string $(0, \ldots, 0)$. Then, $B_i(v)$ contains all nodes having exactly i 1's in their labels. Therefore, the number of neighbors of a node $u \in B_i(v)$ in the proceeding set $B_{i-1}(v)$ is exactly i – those labels obtained by flipping one of the 1's in u's label to a zero. Define $L_i(v) \subseteq B_i(v)$, for $1 \leq i \leq 5$ as follows: $L_1(v)$ is the set of free neighbors of v, $L_2(v)$ the set of covered neighbors of $L_1(v)$, $L_3(v)$ the set of free neighbors of $L_2(v)$, $L_4(v)$ is the set of covered neighbors of $L_3(v)$, and $L_5(v)$ the set of free neighbors of $L_4(v)$. Furthermore, define $N_i(v) \subseteq B_i(v)$, for $2 \leq i \leq 5$ as follows: $N_2(v)$ is the set of neighbors of $L_1(v)$, $N_3(v)$ is the set of free neighbors of $N_2(v)$, $N_4(v)$ is the set of *covered* neighbors of $N_3(v)$ and $N_5(v)$ is the set of free neighbors of $N_4(v)$. Note that $|N_i(v)| \geq |L_i(v)|$ for $i = 2, 3, 4$.

LEMMA 3.3. *Assume that at least half the neighbors of each node are covered. Let v be a vertex and let i, j be such that $1 \leq i \leq \log k$ and $i + j \leq \log k$. Suppose*

$$\frac{k}{2^{i+1}} \leq |N_F(v)| < \frac{k}{2^i}$$

and

$$|L_3(v)| \leq \frac{k^3}{2^{2i+j}}.$$

Then, after $O(\frac{n}{2^j})$ steps, the number of free neighbors of v must be smaller than $\frac{k}{2^{i+1}}$ with high probability.

PROOF. Recall that for every l, a node in $L_{l+1}(v)$ can have at most $l + 1$ neighbors in $L_l(v)$. By this property, we have that $\sum_{u \in L_2(v)} |N_F(u)| \leq 3|L_3(v)| + |L_1(v)|k$, since we count each vertex of $L_3(v)$ at most three times. Since $i + j \leq \log k$ and $|L_1(v)| \leq \frac{k}{2^i}$, we get that $|L_1(v)|k \leq \frac{k^3}{2^{2i+j}}$, and thus, $\sum_{u \in L_2(v)} |N_F(u)| \leq \frac{k^3}{2^{2i+j-2}}$. Furthermore, since each vertex has at least $k/2$ covered neighbors, and each vertex of $L_2(v)$ has at most two neighbors in $L_1(v)$ we have that $|L_2(v)| \geq |L_1(v)|\frac{k}{4} \geq \frac{k^2}{2^{i+3}}$.

We now lower bound the probability of covering a vertex of $L_1(v)$ as long as $|L_1(v)| \geq \frac{k}{2^{i+1}}$. This probability is at least

$$\begin{aligned} \sum_{u \in L_2(v)} \frac{1}{n} \cdot \frac{1}{|N_F(u)|} &\geq \frac{1}{n} |L_2(v)|^2 \frac{1}{\sum_{u \in L_2(v)} |N_F(u)|} \\ &\geq \frac{2^{2i+j-2}|L_2(v)|^2}{k^3 n} \geq \frac{k 2^{j-8}}{n}, \end{aligned}$$

where the first inequality follows from the convexity of the function $f(x) = 1/x$. The proof now follows by the same arguments given in Lemma 3.2.

□

3.3 Establishing a linear bound

As before, we will have $\log k+1$ phases numbered $0, 1, \ldots,$ $\log k$, and will prove that at the end of the i^{th} phase, the number of free neighbors of each node is at most $\frac{k}{2^{i+1}}$, with high probability. Phase 0 is as before, and takes $O(n)$ steps. In each of the remaining phases, we will execute a number of iterations whose purpose is to "thin down" $L_3(v)$, for each node v. This time around, $L_4(v)$ acts as a funnel, and since we are dealing with an aggregate of nodes, the funnel is much larger and so the number of steps required is fewer than in Lemma 3.2. Next, we will "thin down" $L_1(v)$. By Lemma 3.3, this also takes fewer steps than before. The number of iterations executed in phase i is $t_i = \min\{i, \log k - i\}$. Each iteration within phase i consists of $O(n/2^i)$ steps.

LEMMA 3.4. *By the end of iteration j within phase i, with high probability, for each node v with $|N_F(v)| \geq k/2^{i+1}$,*

$$|L_3(v)| \leq \frac{k^3}{2^{2i+j}}.$$

PROOF. The proof is similar to that of Lemma 3.2, with $L_4(v)$ acting as a funnel to cover nodes in $L_3(v)$.

We will actually prover a stronger claim: By the end of iteration j within phase i, with high probability, for each node v with $|N_F(v)| \geq k/2^{i+1}$,

$$|N_3(v)| \leq \frac{k^3}{2^{2i+j}}.$$

Clearly, since $|L_3(v)| \leq |N_3(v)|$, the lemma follows.

Assume that $|N_3(v)| > \frac{k^3}{2^{2i+j}}$ in the beginning of iteration j (otherwise we are done with $N_3(v)$ for iteration j).

Recall that for every $u \in N_4(v)$, $d_u(N_3) \leq 4$ and that for every $u \in N_3(v), |N_C(u)| \geq k/2$, and on the other hand $d_u(N_2) \leq 3$. Thus, $|N_4(v)| \geq |N_3(v)|(k/8-3) \geq |N_3(v)|k/16$ for sufficiently large k. As long as $|N_3(v)| > \frac{k^3}{2^{2i+j}}$, the probability of covering a vertex in $N_3(v)$ is at least

$$\sum_{u \in N_4(v)} \frac{1}{n}\frac{1}{|N_F(u)|} \geq \frac{2^i}{kn}|N_4(v)| \geq \frac{2^i}{8n}|N_3(v)| > \frac{k^3}{2^{i+j+4}n}.$$

We note that the quantity $i + j$ is not increasing with i since $i + j \leq \log k$. Assume that iteration j consists of $cn/2^i$ steps for c large enough. If $|N_3(v)| > \frac{k^3}{2^{2i+j}}$, the number of vertices of $N_3(v)$ covered in iteration j is a random variable dominating the binomial random variable $X \sim B(\frac{cn}{2^i}, \frac{k^3}{2^{i+j+4}n})$. By Lemma 3.1,

$$\begin{aligned} prob(X < \frac{|N_3(v)|}{2}) &\leq prob(X < \frac{k^3}{2^{2i+j}}) \\ &= prob(X < \frac{ck^3}{2^{2i+j+4}} - \frac{k^3}{2^{2i+j}}(\frac{c}{16} - 1)) \\ &< e^{-\Omega(k^3/2^{2i+j})} \leq e^{-\Omega(k)}, \end{aligned}$$

where the last two inequalities follow from the fact that c is large enough, and that $i + j \leq \log k$. We now use the union bound over all 2^k vertices, and all $O(\log^2 k)$ iterations, and we get that for every v, by the end of iteration j, $|N_3(v)| \leq \frac{k^3}{2^{2i+j}}$ with high probability.

□

After the t_i iterations in phase i, we execute $O(\frac{n}{2^{t_i}})$ more steps. By Lemma 3.3, these suffice to ensure with high probability that $|N_F(v)| \leq \frac{k}{2^{i+1}}$ for all nodes v.

Finally, since

$$\sum_{i=1}^{\log k} \frac{t_i}{2^i} + \sum_{i=1}^{\log k} \frac{1}{2^{t_i}} \leq \sum_{i=1}^{\log k} \frac{i}{2^i} + 2\sum_{i=1}^{\frac{\log k}{2}} \frac{1}{2^i} = O(1),$$

the total number of steps is $O(n)$, hence proving Theorem 2.1.

3.4 Establishing the Lower Bound

In this section we prove Theorem 2.2. We shall show that there is a constant c such that with high probability each of the first $\lfloor cn \rfloor$ steps covers an uncovered vertex. We will use the following lemma (its proof can be found, e.g., in [2]):

LEMMA 3.5. *Let Z be a random variable with a binomial distribution $Z \sim B(n, p)$. For every $\lambda > 1$, $prob(Z > \lambda np) < (e^{\lambda-1}\lambda^{-\lambda})^{np}$.*

In order to prove Theorem 2.2, we introduce the following definitions. As before, for every $v \in V$ we define $L_1(v) = N_F(v)$ and $L_2(v) = \bigcup_{u \in L_1(v)} N_C(u) - \{v\}$. We say that step i is successful if after step i, for every vertex v, $|L_1(v)| \geq k/2$. The theorem follows from the following claim:

CLAIM 3.6. *There exists a universal constant c, such that for $1 \leq t \leq ck^3$, the first $\frac{tn}{k^3}$ steps are successful with high probability.*

PROOF. The proof is by induction on t. For t a positive integer, let A_t be the event that the first $\frac{tn}{k^3}$ steps are successful. For the base case, we show that A_1 occurs with high probability. In order to cover a vertex of $L_1(v)$, we have to either choose a vertex u of $L_2(v)$, or choose vertex v. Since $|L_2(v) \cup \{v\}| \leq k|L_1(v)| \leq k^2$ we get that the probability of covering a vertex in $L_1(v)$ is at most $\frac{k^2}{n}$. Let X be the random variable that denotes the number $k - |L_1(v)|$ after the first n/k^3 steps. Let Y be a random variable with the binomial distribution, $Y \sim B(\frac{n}{k^3}, \frac{k^2}{n})$. We get that Y dominates X, and by Lemma 3.5,

$$prob(X > k/4) \leq prob(Y > k/4) = e^{-\Omega(k \ln k)}. \quad (1)$$

Therefore, by a union bound, with high probability, for every vertex v, $|L_1(v)| > k/2$.

We now prove the inductive step: we assume that A_t occurs with high probability, and show that if $t \leq k^3$, then A_{t+1} also occurs with high probability. Let $A'_t(v)$ be the event that after the first $\frac{tn}{k^3}$ steps, $|L_1(v)| \geq 3k/4$. We first show that if A_t occurs, then we can actually make the stronger claim that for every vertex v, $A'_t(v)$ occurs with high probability. We then use a similar argument as above to show that, with high probability, in the last n/k^3 steps, for any v, at most $k/4$ additional vertices in $L_1(v)$ get covered.

To show that $A'_t(v)$ occurs with high probability, note that $\Pr[\overline{A'_t(v)}] \leq \Pr[\overline{A'_t(v)} \cap A_t] + \Pr[\overline{A_t}]$, and thus, by the inductive hypothesis, we only need to show that the probability of $\overline{A'_t(v)} \cap A_t$ is inverse polynomial in n. Thus, we focus on

the first tn/k^3 steps, and only consider the case where for every $u \in L_2(v)$, $|L_1(u)| \geq k/2$.

Since each vertex in $L_2(v)$ has at most two neighbors in $L_1(u)$, in each of the first tn/k^3 steps, the probability of covering a vertex of $L_1(v)$ is at most $\frac{4}{kn}|L_2(v)| \leq \frac{4k}{n}$. Let X_1 be the random variable that denotes the number $k - |L_1(v)|$ after the first tn/k^3 steps. Let Y_1 be a random variable with the binomial distribution, $Y_1 \sim B(\frac{tn}{k^3}, \frac{4k}{n})$. We get that Y_1 dominates X_1, and by Lemma 3.5,

$$prob(X_1 > k/4) \leq prob(Y_1 > k/4) = e^{-\Omega(k \ln(k^3/t))},$$

as long as $t < ck^3$ for sufficiently small (but constant) c. If c is small enough, then we get by a union bound that for every vertex v, $X_1(v) \leq k/4$.

We are now left with the last n/k^3 steps. We define $X = k - |L_1(v)| - X_1$ to be the number of vertices removed from $L_1(v)$ in the last n/k^3 steps, and define Y as before, i.e. $Y \sim B(\frac{n}{k^3}, \frac{k^2}{n})$. Then the situation is exactly as in the base case, and therefore Equation (1) holds, and by a union bound we can assume that for every vertex v, $X < k/4$, and thus, $|L_1(v)| > k/2$. This completes the proof. □

4. EXTENSIONS OF THE PROCESS

In this section we introduce and analyze a variant of Process 1, and give some upper bounds on the covering time of the processes in different graphs. We first show that our proof holds for every d-regular graph. We have the following theorem:

THEOREM 4.1. *Let $G = (V, E)$ be a d-regular graph with n nodes. With high probability, after $O(n(1 + \frac{\log n \cdot \log d}{d}))$ steps of Process 1, all nodes are covered.*

PROOF. We first note, that in the proof of Lemma 3.2 we did not use the fact that graph is the hypercube, but simply the fact that the graph is k-regular for $k = \log n$. Thus, we only have to show that the same holds for d-regular graphs for $d \neq \log n$.

The proof has the same flavor as Lemma 3.2. We have $\log d$ phases. After phase i, we will have at most $\frac{d}{2^i}$ uncovered neighbors for every vertex v. Since the coupon collector process is dominated by Process 1, it is easy to see that by using $\Theta(n + n \log n \frac{\log d}{d})$ steps for phase 0, we have that each vertex has at least $d/2$ covered neighbors with high probability.

As long as $|L_1| \geq d/2^{i+1}$, the probability of covering a vertex in L_1 is at least

$$\sum_{u \in L_2} \frac{1}{n} \cdot d_u(L_1) \frac{2^i}{d} = \frac{2^i}{nd} \sum_{v \in L_1} |N_C(v)| \geq \frac{2^{i-1}}{n} |L_1| \geq \frac{d}{4n}.$$

Consider a sequence of cn steps of the algorithm where c will be determined later, and let X be a random variable, where $X \sim B(cn, \frac{d}{4n})$. As long as $|L_1| \geq d/2^{i+1}$, the number of vertices of L_1 covered by the process stochastically dominates X, that is, the probability that after cn steps we still have $|L_1| \geq d/2^{i+1}$ is at most the probability that $X < d/2^{i+1}$. By Lemma 3.1 we get

$$prob(X < \frac{d}{2^{i+1}}) \leq prob(X < \frac{cd}{4} - \frac{d}{4}(c - 1/2^{i-1})) \leq e^{-\Theta(cd)},$$

for $c > \frac{1}{2^{i-2}}$.

If $d/2^i > 10 \log n$, we take $c = \Theta(1/2^i)$, and otherwise we take $c = \Theta(\frac{\log n}{d})$. We thus get that $prob(X < \frac{d}{2^{i+1}})$ is inverse polynomial in n, and by applying a union bound over all $\log d$ phases and all n vertices, we get that with high probability after phase i each vertex has at most $d/2^{i+1}$ uncovered neighbors. The total number of steps we use for all the phases is at most $O(n + n \frac{\log n \cdot \log d}{d})$. □

As a corollary of Theorem 4.1 we get that for every d-regular graph with $d > \log n \cdot \log \log n$, $O(n)$ steps of Process 1 are sufficient to cover all nodes with high probability.

Modify Process 1 so that after picking a random node, v, it only covers a random uncovered neighbor of v. If all neighbors of v are covered, it does nothing. Call this Process 2. In this process, a node has a funnel of size k all the way until it is covered. Hence, phase 0 is unnecessary, and the rest of the proof remains essentially unchanged to show the following theorem:

THEOREM 4.2. *Using Process 2, $O(n)$ steps suffice to cover all nodes of the hypercube with high probability. Furthermore, $O(n + n \frac{\log n \cdot \log d}{d})$ steps are sufficient to cover all nodes of any d-regular graph.*

Next, consider a family of graphs having maximum degree d, minimum degree $\Omega(d)$ and the following additional property: For every $v \in V$, if we conduct Breadth first search from v, and let B_i for $i = 1, 2, \ldots$ denote the levels. Assume that for $i = 2, 3, 4$ and for every $u \in B_i$, the degree of u into B_{i-1} is bounded by a constant. Then, it is easy to see that the proof of Theorem 2.1 goes through, and we get that $O\left(n(\frac{\log n}{d} + 1)\right)$ steps suffice to cover all nodes of the graph with high probability using either Process 1 or Process 2. Since random d-regular graphs satisfy these conditions, we get

THEOREM 4.3. *For a random d-regular graph, the number of steps suffice to cover all nodes of the graph with high probability using either Process 1 or Process 2 is at most $O\left(n(\frac{\log n}{d} + 1)\right)$.*

Consider now the infinite process inferred by the hypercubic hash table, where there is an infinite sequence of arrivals. The following theorem shows that the probability that there never is a difference of more than d between the shallowest and deepest levels of the tree is at least $1 - e^{\Omega(d)}$. This demonstrates that this process is self correcting.

THEOREM 4.4. *The probability that there is never a difference of more than d between the shallowest and deepest levels of the tree in the hypercubic hash table over an infinite sequence of arrivals is at least $1 - e^{\Omega(d)}$*

PROOF. Let A_m denote the event that until step m the difference between the shallowest and deepest levels is at least d. We first note that one can modify the constants c_1, c_2 from Corollaries 2.3 and 2.4 such that the events described in the lemmas happen with probability at most $\frac{1}{m^2}$ where m is the number of steps. Assume that d is much larger than the constants c_1, c_2.

Consider the first $N = 2^{d-c_1}$ steps. By Corollary 2.4, with probability at least $1 - 1/N^2$ the deepest level until that point is at most d. Therefore, $prob(A_1 \cup \ldots \cup A_N) \leq \frac{1}{N^2}$. On

the other hand, by Corollaries 2.3 and 2.4, $prob(A_m) \leq \frac{2}{m^2}$. Therefore,

$$prob(\bigcup_{m>N} A_m) \leq \sum_{m>N} prob(A_m) \leq \sum_{m>N} \frac{2}{m^2} \leq \frac{2}{N}.$$

Since $N = \Theta(2^d)$ the theorem follows. □

5. AN ALTERNATIVE PROOF TO THEOREM 2.1

We now introduce a different proof of Theorem 2.1 using coupling. This proof provides a tighter analysis than the one given in Section 3, but it is not clear how to extend this proof to other cases such as d-regular graphs.

We will prove the theorem for Process 2, but we note that the proof is almost identical for Process 1. Let T_n be the *cover time* of Process 2, that is, the number of steps until every vertex is covered. We shall investigate the probability distribution of T_n.

We prove the Theorem by comparing T_n with the cover times of two other stochastic processes: a *random probing process* with cover time P_n and a *rejection process* with cover time R_n.

By comparing the random probing process with the standard coupon collector process we show that $P_n = O(R_n)$ with high probability. We then show that the rejection process can be viewed as the random probing process slowed down by a constant factor. Finally, we complete the proof with a coupling argument showing that T_n is stochastically smaller than R_n.

The proof is designed for clarity rather than for optimizing the constant factor in the time bound for T_n.

The random probing process is as follows. Initially all vertices are uncovered. At each step a multiset S of k vertices is chosen uniformly at random with replacement. If S contains an uncovered vertex then an occurrence of an uncovered vertex in S is chosen uniformly at random and the vertex becomes covered; otherwise no vertex becomes covered at this step. Define the *cover time* P_n of this process as the number of steps until every vertex is covered.

We shall relate P_n to the standard *coupon collector process*, in which elements are drawn with replacement independently and uniformly at random from an n-set until all elements have been drawn. Let C_n be the number of draws in the coupon collector process.

LEMMA 5.1. *P_n is stochastically smaller than $\frac{C_n+n(k-1)}{k}$*

PROOF. Consider the following method of implementing the random probing process: at each step the vertices in S are inspected sequentially in a random order until an uncovered vertex is found or the set S has been exhausted. Once an uncovered vertex has been found the remaining vertices in S are discarded without being inspected. The number of inspected vertices is distributed as C_n, and the number of discarded vertices is always less than or equal to $n(k-1)$. But dkP_n is equal to the number of inspected vertices plus the number of discarded vertices. □

LEMMA 5.2. *$P(n) < n(\frac{1}{log_2 e} + 2)$ whp*

PROOF. The probability that $C(n) > n(\ln n + k)$ is bounded above by $n(1-1/n)^{n(\ln n+k)}$ which is exponentially small in k. Combining this fact with the above lemma and the fact that $k = \frac{\ln n}{\log_2 e}$ the result follows. □

For any vertex v, let $N(v)$ be the set of neighbors of v in the hypercube. Consider a state S (i.e., a designation of each vertex of the hypercube as covered or uncovered). Let $a(u,S)$ denote the number of uncovered neighbors of u and let $a_2(u,S)$ denote the number of uncovered vertices at distance 2 from u. Define $Cov(v,S)$, the *covering probability* of an uncovered vertex v, as the probability that v will get covered at the next step of the covering process, given that the present state is S. Then $Cov(v,S) = \frac{1}{n}\sum_{u\in N(v)} 1/a(u,S)$.

We now give the definitions and lemmas required for defining and analyzing the rejection process. Let p be a rational number i/n between 0 and 1. A *random p-state* is a state drawn uniformly at random from the set of states in which there are exactly pn uncovered vertices.

LEMMA 5.3. *In a random p-state S, the probability is less than $\frac{1}{n^2}$ that there exists an uncovered vertex u such that $Cov(u,S) < \frac{1-(1-p)^k}{6.4pn}$.*

PROOF. In a random p-state S the random variable $a_2(u,S)$ has a hypergeometric distribution with mean $\binom{k}{2}p$. Using Chvatal's bound on the tail of the hypergeometric distribution [5] the probability that $a_2(u,S) > s\binom{k}{2}p$, where $s > 1$, is less than $(e/s)^{s\binom{k}{2}p}$. Let r be the value of s where this bound is equal to 6^{-k}. Then, by a union bound, with probability $1-4^{-k}$, $a_2(u,S) \leq r\binom{k}{2}p$ for every vertex u. Let v be an uncovered vertex in state S. If $a_2(v,S) \leq rp\binom{k}{2}$ then $\sum_{u\in N(v)} a(u,S) = k + 2a_2(u,S) \leq k + 2rp\binom{k}{2}$ and, by the convexity of the function $1/x$ it follows that $Cov(v,S) \geq \frac{k}{n(1+rp(k-1))}$. Therefore, with probability $(1-4^{-k})$, this inequality holds for every uncovered vertex in S.

We now examine the ratio $\frac{1+rp(k-1)}{1+p(k-1)}$. We shall show that this ratio is less than 3.2. Since the ratio is less than r, we may assume that $r > 3.2$. Since $(e/r))^{r\binom{k}{2}p} = 6^{-k}$, $(r/e)^{r\frac{(k-1)p}{2}} = 6$. Let $a > 0$ be such that $r/e = 6^a$. Then $ae6^a\frac{(k-1)p}{2} = 1$. It follows that $p(k-1) = \frac{2}{ae6^a}$, $2rp(k-1) = 2/a$ and $\frac{1+rp(k-1)}{1+p(k-1)} = \frac{1+2/a}{1+\frac{2}{ae6^a}}$. It is easy to show that this function is bounded over the positive reals, and numerical computation shows that it is less than equal to 3.2 for all positive a.

Next we observe that $\frac{(1-(1-p)^k)(1+p(k-1))}{kp} \leq 2$. Putting the inequalities together, the lemma follows. □

The rejection process is designed so that, at any stage where the number of uncovered vertices is pn, all p-states are equally likely. We shall show by induction over the steps of the process that this property holds (with high probability) and that, when the process is in a p-state S, every uncovered vertex u satisfies $Cov(u,S) < \frac{1-(1-p)^k}{6.4pn}$. The process operates as follows. Initially all vertices of the k-dimensional unit hypercube are uncovered. At each step, let S be the state, let i be the number of uncovered vertices, and let $p = i/n$. A vertex v is chosen at random. If v has at least one uncovered neighbor then an uncovered neighbor u of v is chosen uniformly at random. By induction hypothesis, assume that $Cov(u,S) \geq \frac{1-(1-p)^k}{6.4pn}$. With probability $\frac{1-(1-p)^k}{6.4pnCov(u,S)}$ the step is accepted and u becomes covered. With the complementary probability the step is rejected. Each uncovered vertex has probability $\frac{1-(1-p)^k}{6.4pn}$ of becoming covered at this step.

Since, at each step, all uncovered vertices are equally likely to become covered, it follows that, any stage where pn vertices are uncovered, the state is a random p-state, as was required for the induction in the above analysis.

Next we compare the rejection process with the random probing process and with the original covering process. In a p-state the random probing process has probability $1-(1-p)^k$ of covering a vertex, whereas the rejection process has probability $\frac{1}{6.4}(1-(1-p)^k)$ of covering a vertex. Thus the progress of the rejection process is stochastically the same as that of a process which, at each step, tosses a coin with probability of heads $1/6.4$ and, if heads comes up, executes a step of the random probing process. It follows that the number of steps of the rejection process is less than $6.5n(\frac{1}{log_2 e}+2)$ with high probability.

Finally, we observe that the number of steps in the original covering process is stochastically smaller than the number of steps in the rejection process. To see this we use a coupling argument. Assume that, at each step, the two processes select the same random vertex, and inspect its neighbors in the same order to select an uncovered vertex if one exists. Then, step by step, the set of vertices covered by the rejection process is a subset of the set of vertices covered by the covering process.

Summing up, we have:

THEOREM 5.4. *$T_n < 6.5n(\frac{1}{log_2 e}+2)$ with high probability.*

6. EXPERIMENTAL RESULTS AND A DELETION PROCESS

We ran some simulations to study the performance of Processes 1 and 2 in practice. As seen in Figure 2, in practice the number of steps required to cover all nodes of the hypercube is bounded by $2n$. Furthermore, it is not clear cut, but it seems that the constant in Process 1 is slightly better than the constant in Process 2.

Next, we define a leaf deletion process that is in the spirit of our leaf addition process: suppose leaf i needs to be deleted. Consider the leaves that i points to and among the ones at the deepest level pick a random one, say j. It is easy to see that the sibling of j will also be a leaf, say k. Now, the processor at j takes over i's job (the processor at i is relieved), and the processor at k takes over j and k's job and sits at the parent of j and k.

We do not have a good analysis of this deletion process at present, but report the following experiments which suggest that it should also perform well. The first set of experiments start with complete binary trees of depth 20, and at each step performing a delete at a leaf which is chosen uniformly from the set of leaves. It was observed that the difference in depth of the deepest and shallowest leaves was bounded by 4 throughout the process.

The next set of experiments involved starting with a complete binary tree of depth 10, and adding 10,000 more nodes as follows: pick a leaf, and with probability 0.5 declare it a permanent leaf, and with probability 0.5 add two new children to it. In the resulting tree, the difference between depths of deepest and shallowest leaves was 20. Next, we performed random deletes on this tree. After 8,000 steps, the difference went down to 10. We observed that the difference was *almost* monotonically decreasing, in the sense that it increased by only 1 a few times (thus, it would go from 14 to 15, before going to 13, but not from 14 to 15 and then to 16, before going to 13).

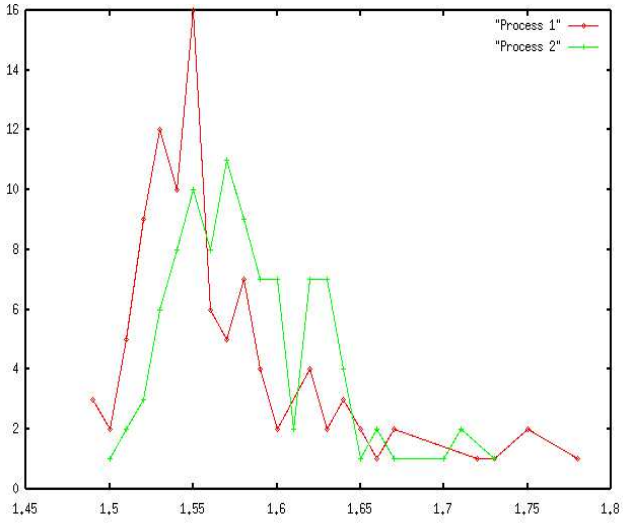

Figure 2: The distribution of the constants d_1, d_2 such that Processes 1 and 2 respectively cover all nodes after $d_1 n$ and $d_2 n$ steps respectively. We preformed 100 experiments on the 25-dimensional hypercube. The x-axis is the constants (d_1 or d_2), and the y-axis refers to how many experiment each constant appeared in. Clearly, both constants are always smaller than 2, and Process 1 is more concentrated than Process 2, and is usually faster.

7. REFERENCES

[1] N. Alon. personal communication, 2002.

[2] N. Alon and J. H. Spencer. *The Probabilistic Method*. John Wiley and Sons, Inc., 2000.

[3] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the 26th ACM Symposium on Theory of Computing*, pages 593–602, 1994.

[4] A. D. Joseph B. Y. Zhao, J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, 2001.

[5] V. Chvatal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25:285–287, 1979.

[6] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, 2002.

[7] R. Karp, M. Luby, and F. Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, 1992.

[8] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[9] M. Mitzenmacher, A. Richa, and R. Sitaraman. The power of two random choices: A survey of the

techniques and results. In P. Pardalos, S. Rajasekaran, and J. Rolim, editors, *Handbook of Randomized Computing*. Kluwer Press, to appear.

[10] C. Greg Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[11] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[12] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.