Ideal Load Balancing Techniques in Structured Peer-to-Peer Networks (#424)

***Abstract*— Peer-to-peer(P2P) networks have grown in popularity in recent years. One of the typical applications of P2P networks is file-sharing. Effectively load balancing in such applications is important since the distribution of the number of requests for individual files can be heavily skewed.**

We consider a file-sharing scenario in a structured Peer-to-Peer (P2P) network that implements a distributed hash table (DHT), such as CAN, Chord or Tapestry. In the basic design of these networks each file is stored at a single node which will become a hotspot if the file is popular. In this paper we present some novel caching/file-replication techniques that will balance the load. The central problem addressed is what nodes should cache the file in order to achieve (close to) perfect balancing. We show that intuitive solutions such as replication at neighbors or along the search path do not have this property. However, we can construct Tapestry-type of networks with randomized routing tables in which replication at nodes with longest common prefix achieves provably ideal load-balancing. We extend this idea to construct a simple caching strategy that adapts to fluctuating loads.

I. Introduction

The popularization of peer-to-peer (P2P) file-sharing applications, such as Kazaa and Gnutella, have brought much attention to P2P networks. We consider the file-sharing scenario in structured P2P networks, such as CAN [6], Chord [7], Pastry[4], or Tapestry [8], that implement distributed hash tables (DHT). In these networks, publishing a file is to insert the file into a hash table, which is distributed in the sense that pieces of it are stored separately in a large number of nodes. Searching for a file is to obtain the hash value of the file or of its metadata and to make a query using the hash value. With specially chosen network structure, the routing tables can be set up at the time of network construction without the help of a routing protocol. The query is routed with the file hash value as the destination. This structured approach overcomes the limitation of existing unstructured file-publication approaches, which require either query flooding or establishing file directories.

Effectively load balancing is important in any large-scale file-sharing applications since the distribution of the number of requests for individual files can be heavily skewed. There have been many widely reported incidences, often associated with the occurrence of some events, of extremely large number requests for certain file or web content overburdening the content server. Even under normal circumstances, it is conceivable that file popularity, measured in the number of requests, follows a heavy-tail distribution, given the ubiquity of this type of distributions[1]. As a consequence of the heavy-tail distribution, there is a non-negligible chance that some files are extremely popular. The servers for the popular files can be many times overloaded above their capacity.

We stress that proper load balancing is especially important for a P2P network constructed from voluntary participation. If participation in the network overloads some nodes, then they may stop serving popular files, which, in turn, increases the chance for the remaining nodes to be overloaded. This suggests that, while it is difficult to grow a P2P network, collapsing a widely used network might be quite quick and easy. One interesting thing about the DHT-based file-sharing network (as compared to the query-flooding-based network) is that, if a node realizes that it is carrying a popular file, it might quit the network and rejoin with a different node ID, hoping that it won't carry any popular file this time. As a result, it may be difficult to find and successfully download popular files as nodes dodges them. Furthermore, the frequent leave and join by nodes may seriously undermine the stability of the network because the protocols used for constructing and maintaining the structure of the network are not given enough time to do their job. Even though we cannot fully anticipate the dynamics of the user behavior, it does seem critical that structured file-sharing networks need to be engineered carefully with proper load balancing[2].

The structured network can store either files themselves or references to files, such as the IP address of the host that contains a file and metadata about the file. In this paper, whenever we mention the term *file*, we could mean either the file itself or the file reference. To search or publish a file, a hash function is first applied to the file and a file ID is returned. Because the links and routing tables are set up at the time of network construction, any node or file can be located by routing a query message using the node or the file ID as the destination. Thus, the combination of hashing and routing together allows the identification and location of the node that stores the file. Because of the involvement of routing, the structured networks such as CAN, Chord and Tapestry are said to implement a distributed DHT.

Even though hashing is distributed, with respect to a particular file, the basic version of CAN, Chord or Tapestry is each a centralized system. Each file is always stored in one node. This poses the classical problems about a centralized system: *single point of failure*, that is, when a node fails, files on the node will no longer be accessible; and *hot spot*, that is, all requests for a particular file are directed to one node, causing overload to that node or the network paths leading to that node. The basic DHT using a single hash function does not exploit the vast capacity and high redundancy of a typical P2P

[1]In a heavy-tail distribution, the probability mass function (pmf) decays as a power function.

[2]Unless other incentive structures are set up so that the owners of overloaded nodes are compensated for losing control of their computers.

network, which has a large number of users, lots of storage space, processing power and communication bandwidth, and potentially has many replicas for many files.

Large file-sharing P2P networks (i) should have build-in resiliency or fault tolerance to combat the problem of single point of failure, (ii) should have load balancing mechanism to solve the hot-spot problem, and (iii) it is also desirable to replicate the file content to increase the lifetime of each file. In this paper, we will mainly consider file replication for load-balancing purpose. The central problem addressed is what nodes should cache the file in order to achieve (close to) perfect balancing. We show that intuitive solutions such as replication at neighbors or along the search path do not necessarily have this property. However, we can construct Tapestry-type of networks with randomized routing tables in which replication at nodes with longest common prefix achieves provably ideal load-balancing. We extend this idea to construct a simple caching strategy that adapts to fluctuating loads.

The file-replication strategies proposed in CAN, Chord, Pastry, Tapestry and some other studies [5] can be summarized into three categories, (i) caching (ii) replication at neighbors, and (iii) replication with multiple hash functions. Caching a file can be done when the file is first published, at nodes along the route of the publishing message, and/or, when the file is requested, at one or more nodes along the route of the query message. In the second approach above, when a node is overloaded with the requests to some file, it replicates the file at its neighbors, i.e., the nodes to which it has direct (virtual) links.

Each of these strategies has its advantages and disadvantages. Caching is the simplest approach and it can improve the response time of the queries if done properly. However, a simple caching algorithm cannot be a complete solution to the load balancing problem, because even a good cache hit ratio, say 80%, still leaves 20% of the requests going to the original node for the file, which may overload the node by many times. Replication-at-neighbors does not have cache miss problem, if the file is replicated at all neighbors of the original server. However, a bit surprisingly, even in these very "uniform" networks, the load to each of the neighbors may not evenly distributed. In general, it is difficult to achieve guaranteed load balancing with this approach because the assignment of requests to nodes depends on many factors and is not tightly controlled. Furthermore, in some networks, such as CAN, neighbors of a node are actually close to each other in the name space of nodes (rather than in physical sense). Even after the nodal hot spot is removed, the routing hot spot may still remain. In this paper, we combine caching and replication-at-neighbors in a way that load balancing is guaranteed in some particular Pastry/Tapestry-type networks, as long as the request pattern is uniform. We will still loosely call it caching.

The main advantage of replication with hash functions is that, with the help of uniform hash functions, copies of the file are uniformly distributed over the network, and with uniform use of the hash functions, file requests are also uniformly distributed over the set of nodes that contain the file. However, it is not easy to decide how many hash functions are needed. In a separate study, we develop mechanisms to address this problem, but the price to pay is increased response time for queries.

II. Caching in Pastry/Tapestry-Type Networks

A. Routing Table and Query Routing in General Pastry Network

We consider Pastry-type networks, in which Tapestry is one example. To illustrate the basic idea about the network construction and routing, let us first consider a network with n nodes, numbered $0, 1, ..., n-1$, and the size of the name space is also n. Let $n = 2^e$, where e is a natural number. Each file is mapped into a key value in $\{0, 1, ..., n-1\}$ through a uniform hash function, and the key value indicates the ID of the node where the file is stored. The node ID's and the file keys can all be expressed as binary numbers. When a file is requested, the query for the file is routed from the starting node to the node whose ID is equal to the file key. Suppose the initial node ID is $a_{e-1}a_{e-2}...a_0$ and the file key is $b_{e-1}b_{e-2}...b_0$, where $a_i, b_i \in \{0, 1\}$. In Pastry or Tapestry, routing of the query from node $a_{e-1}a_{e-2}...a_0$ to node $b_{e-1}b_{e-2}...b_0$ can be viewed as changing the former number to the latter one bit at each hop. More specifically, by moving to the i^{th} hop, $i = 1, 2, ..., e$, a_{e-i} is changed into b_{e-i} if they are not the same. This immediately implies that the length of the query path is $\log_2 n$.

The construction of the Pastry's network and the routing table at each node determine the above routing behavior. Table I shows the routing table of node $a_{e-1}a_{e-2}...a_0$. The search key is checked against the routing table when making the routing decision. The first column is called the *level*, which is the position of the digit currently under consideration. If the query message has traversed i hops (including the current node), then the i^{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's of the next hops, i.e., the neighboring nodes. At each level, say level i, each next-hop node must satisfy the following requirement: (i) the i^{th} digit of the next-hop node must match the value in column 2, which will be the value of the i^{th} digit of the search key when a query arrives at the node after traversing i hops (inclusive); 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 neighbors. Each "$*$" in the table is a wild card, which can be either 0 or 1.

Each particular choice for the wild cards in the routing table defines a specific type of the Pastry networks, which all have the distinguishing changing-one-bit-at-a-time routing behavior, but other than that, are very different networks with distinct properties. For each routing table entry, the original Pastry[4] chooses an eligible next-hop node that is the closest to the current node, where the distance may be measured in either the round-trip time (RTT) or some other generalized notion. In the original Pastry network, each file is also replicated and

TABLE I

GENERAL ROUTING TABLE FOR NODE $a_{e-1}a_{e-2}...a_0$

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

TABLE II

OUR ROUTING TABLE FOR NODE $a_{e-1}a_{e-2}...a_0$

level/digit	value	next hop
1	0	$0a_{e-2}...a_0$
	1	$1a_{e-2}...a_0$
2	0	$a_{e-1}0a_{e-3}...a_0$
	1	$a_{e-1}1a_{e-3}...a_0$
...	...	...
e-1	0	$a_{e-1}a_{e-2}...a_20a_0$
	1	$a_{e-1}a_{e-2}...a_21a_0$
e	0	$a_{e-1}a_{e-2}...a_10$
	1	$a_{e-1}a_{e-2}...a_11$

cached at different nodes as we will do. The particular choices of the neighboring nodes and of the cache locations allow Pastry to achieve optimal access delay. This is in contrast to our goal of achieving perfect load balancing by appropriate network construction and cache location selection. Tapestry[8] is closely related to the original Pastry and also tries to choose the closest eligible node as the next-hop node for each routing table entry. However, due to the particular incremental way of constructing the Tapestry network, each routing table entry settles with a *random* and close node but not necessarily the closest next-hop node. One of the goals of Tapestry is to reduce the query delay by constructing a so-called topology-aware overlay network, where the overlay neighbors are actually close to each other in the physical underlay network (the IP network). Other considerations such as allowing dynamic growth of the network in completely distributed fashion dictate the particular outcome of routing table entries.

B. Our Choice of Routing Table and Routing Scheme

We now specify our choice for the routing table entries, or equivalently, the set of neighboring nodes. The wild cards in the table are chosen to match the values of the current node ID at the corresponding digit. The result is that each next-hop node matches the current node at all but at most one bit positions. The special bit that can potentially be changed corresponds to the level of the routing table entry. The resulting routing table is shown in Table II. Note that at each level, one of the two entries must match the current node completely. During a route lookup, if the next-hop node is identical to the current node, we then move to the next digit and check the corresponding entry in the next level. This is equivalent to the following routing rule: at each node s and for the query destination d, the first bit position that s and d have different values determines the level. Then, choose the entry corresponding to the value of d at that bit position for the next-hop node.

We note that this particular construction of the routing table clearly requires that every position of the name space is occupied by a node, which is our current assumption. In Section VI, we will remove this assumption and construct a network with similar desirable properties of our current network. We also note in passing that there is a simple work-around with the above assumption. Suppose the name space size is 2^m and the number of nodes is 2^e, with $m > e$. We can insist on naming each node with $m - e$ trailing zeros. As a result, every name of the form $*...*0...0$ with $m - e$ trailing zeros has a corresponding node. The routing table is constructed using only the first e bits, which is the effective ID for a node.

C. Relationship between Pastry and Chord

From the routing table II, we see that the distances (differences in node ID's) to the node's neighbors increase by a factor of 2. To see, suppose the current node is $a_{e-1}a_{e-2}...a_i...a_0$. At each level, one of the next-hop node is in fact the current node. The corresponding entry can be deleted from the routing table. At the $(e-i)^{th}$ level, the remaining next-hop node is either $a_{e-1}a_{e-2}...0...a_0$ when $a_{e-i} = 1$ or $a_{e-1}a_{e-2}...1...a_0$ when $a_i = 0$, with the 0 or 1 at $(e-i)^{th}$ digit from the left. In either case, the absolute difference between the current node and the next-hop node is 2^i. In this case, Pastry essentially turns into Chord, with the exception that, in Chord, all neighbors are on one side of the current node in the name space, and in Pastry, the neighbors may be on both sides the current node. For instance, suppose $n = 16$ and the current node is 1010 in binary (or 10 in decimal). Its neighbors in Chord are 1011(11), 1100 (12), 1110 (14) and 0010(2). Its neighbors in Pastry are 1011(11), 1000(8), 1110(14) and 0010(2). There is no fundamental difference between Chord and Pastry if the routing tables of the latter is specified as in Table II. We suspect that the load-balancing properties of our network also largely apply to Chord.

III. REPLICATION AT NODES WITH LONGEST COMMON PREFIX

A. What's Wrong with Replication-at-Neighbors?

Without loss of generality, let us focus on queries for file key 00...0(0), which is stored at the node 00...0(0). Consider the following simple file replication strategy. When node 0 is overloaded, it replicates the file at all its upstream network neighbors, i.e., all nodes that has node 0 as a next-hop node. They are nodes 0...01(1), 0...010(2), 0...0100(4), ..., and 10...0(2^{e-1}). We will show that the loads to these neighbors are very uneven.

Suppose the starting node of each query for file key 0 is uniformly randomly chosen from the n nodes. We ask: what is the chance that the query passes through node 0...01 before ending at node 0? The answer is 1/2. This is because

the probability that a random node has a 1 in the rightmost position, i.e., is of the form $*...*1$, is exactly 1/2. Starting from one such node, with one bit matching the destination at each hop during the routing, the query will reach node 0...01. Queries starting from any node of the form $*...*0$ will not go through node 0...01.

The same argument easily applies to neighboring nodes with ID 2^i in decimal, $i = 1, 2, ..., e-1$. A query will go through node 2^i if and only if it originates at a node of the form $*...*10...0$ where 1 is in the $(i+1)^{th}$ position counting rom the right. The probability of the event is $1/2^{i+1}$. Hence, the load to each of the e neighbors are drastically different, which violates the basic load balancing requirement.

B. Replication at Nodes with Longest Common Prefix

Comparing two of node 0's upstream neighbors, we see that node 1 receives half of all the requests, but node 10...0 receives $1/2^e$ of all requests. This suggests that replicating the file at node 10...0 does not help much in reducing the load to node 0. Upon closer examination, a node that shares longer prefix with the search key receivers more requests than a node that shares shorter prefix. Given a node s and a search key t, $s, t \in \{0, 1, ..., n-1\}$, let $l(s,t)$ be the length of the longest common prefix between s and t.

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

Proof: A query goes through node s if and only if the starting node of the query shares the $e - l(s,t)$-suffix with node s. There are exactly $2^{l(s,t)}$ such starting nodes. Hence, the probability one such node is selected is $2^{l(s,t)}/2^e$, where $2^e = n$ is the total number nodes in the network. ■

One corollary of the lemma is that, given two node $s_1, s_2 \in \{0, 1, ..., n-1\}$ 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 we wish to reduce the load to node t, s_1 is the preferred location for file replication. This suggests the following replication scheme for the file with key t: Replicate the file at nodes in decreasing order of $l(s,t)$. We call this scheme *Longest-Common-Prefix-Based Replication* (LCP-replication). We call a node that contains the file *cache node*.

As an example, suppose $t = 0$. When node 0 is overloaded, the file will be replicated at node 0...01. Now the file is cached at nodes of the form $0...0*$. If these nodes are still overloaded, the file will be replicated at node 0...010 and 0...011. As a result, the files will be cached at nodes of the form $0...0**$. In general, after p replication-steps, each of the nodes of the form $0...0*...*$, where the last p digits are wild cards, has a copy of the file.

With respect to each query destination (or equivalently, each file), it is helpful to consider the network as a tree rooted at the destination of the query, $t = 0$, in this case. We define *levels* for the tree as follows, and call the tree *prefix tree*. Level 0 has the root node. Level i contains all nodes s, with $l(s,t) = e-i$, for $i = 0, 1, 2, ..., e$. An edge from level i to level j, where $i < j$, attaches a level-i node of the form $0...01a_{i-2}...a_0$ to a level-j node of the form $0...010...01a_{i-2}...a_0$, where the first 1 from the left occurs after $e-j$ 0's. In other words, following the edge from the level-j node to the level-i node corresponds to flipping the first 1 to 0. Figure 1 shows one example of the prefix tree for the case of $n = 16$. Note that a node at level i has exactly one child at each level $j > i$. The last level, level 4, is not shown in the figure. Each visible node has one incomplete edge that is supposed to be attached to one unseen level 4 node.

For the moment, assume the root node is the only node that contains the corresponding file. Let $p(s)$ be the probability that a query from a uniform random node arrives at node s. The prefix tree has the following properties: (i) by Lemma 1, for any node s at level i, $p(s) = \frac{1}{2^i}$; (ii) for any node s, $p(s) = \sum_{v \in \Pi(s)-\{s\}} p(v)$, where $\Pi(s)$ is the set of nodes on the subtree rooted at node s; and (iii) for any node s at a fixed level and its (only) child at one level down, denoted by $\sigma(s)$, $p(s) = 2p(\sigma(s))$.

With the prefix tree, we see that the LCP-replication algorithm progressively replicates the file at nodes level by level. We will show that the algorithm achieves perfect load balancing.

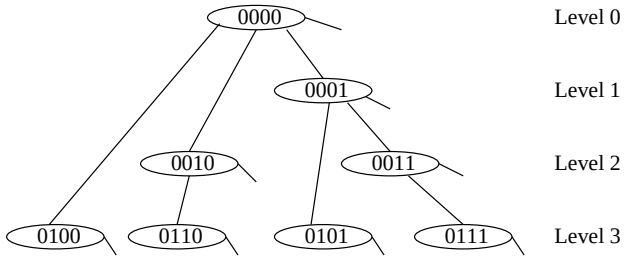

Fig. 1. Prefi x tree for the case of $n = 16$

Lemma 2: Suppose queries for t originates from random nodes, uniformly distributed on $\{0, 1, ..., n-1\}$. After p replication steps, $p = 1, 2, ..., e$, the load to each node that contains a copy of the file is $\frac{1}{2^p}$.

Proof: Without the loss of generality, let us assume the destination is $t = 0$. If this is not true, we can rename each node ID or file key, say id, to $id - t$. We will prove the lemma by induction. After the first replication, node 0...01 has a copy of the file. By Lemma 1, this node receives $\frac{1}{2}$ fraction of the queries for 0. The other half of the queries end at node 0.

Now, suppose the lemma is true for p. The files are copied to nodes of the form $0...0*...*$, where the last p digits are wild cards. At the $(p+1)^{th}$ step, the file is replicated to nodes of the form $0...01*...*$, where the first 1 from the left occurs at the $(p+1)^{th}$ position from the right. Exactly 2^p new nodes are added as cache nodes. Again by Lemma 1, each of these 2^p new nodes receive $\frac{1}{2^{e-(p+1)}}$ fraction of the queries. Let us consider an arbitrary new node added in the $(p+1)^{th}$ step, say node $0...01a_{p-1}a_{p-2}...a_0$. The queries now received and served by the new node $0...01a_{p-1}a_{p-2}...a_0$ was originally received and served by node $0...0a_{p-1}a_{p-2}...a_0$ just before the $(p+1)^{th}$ replication. Hence, after the $(p+1)^{th}$ replication, the load to node $0...0a_{p-1}a_{p-2}...a_0$ is reduced by $\frac{1}{2^{e-(p+1)}}$. By the induction hypothesis, the load to node $0...0a_{p-1}a_{p-2}...a_0$ after the $(p+1)^{th}$ replication must be $\frac{1}{2^{e-(p)}} - \frac{1}{2^{e-(p+1)}} = \frac{1}{2^{e-(p+1)}}$.

■

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) the load to each these nodes is identical, which is the basic requirement for load-balancing, and (iii) after each step of replication, the load to each node is reduced by half.

IV. Automatic Load Balancing by Caching

The LCP-replication algorithm is not automatically a distributed algorithm. Imagine how a group of 2^p nodes that contain the file decide to replicate it in another 2^p nodes. Moreover, it is a slotted algorithm, where a group of nodes operate in a synchronized fashion. We propose the following adaptive and asynchronous algorithm, which is essentially a caching scheme, that automatically achieves load-balancing in the spirit of LCP-replication.

Suppose, for each file f, each node keeps a threshold, θ_f. The node measures the rate of requests for file f, denoted by $r(f)$. If $r(f) > \frac{1}{2}\theta_f + \epsilon$, then the node will cache a copy of the file. When the rate drops to the point that $r(f) < \frac{1}{2}\theta_f - \epsilon$, the node removes that file from its memory.

To understand the above algorithm, let us consider the following idealized situation. Suppose each node in the network sends queries for the file at rate λ_f. Hence, the total query rate is $n\lambda_f$. Suppose $n\lambda_f/\theta_f = 2^p$, for some $p \in \{0, 1, ..., e-1\}$. Let us also modified the algorithm as follows: at each node, if $r(f) > \frac{1}{2}\theta_f$, then the node will cache a copy of the file; if $r(f) \leq \frac{1}{2}\theta_f$, the node removes that file from its memory. Let us consider the static outcome of the algorithm. Clearly, one possible outcome is that the file is replicated at nodes $0...0a_{p-1}a_{p-2}a_0$, where $a_i \in \{0, 1\}$ for $i = 0, 1, ..., p-1$, and nowhere else. Each of these cache nodes receives and serves the queries at exactly rate θ_f. Each of the other nodes receives the queries at rate no greater than $\frac{1}{2}\theta_f$ and will not cache of copy of the file. This is exactly the outcome of the LCP-replication algorithm, which is what we wish to see. One question is: does any other outcome exist? The following lemma shows that the answer is no.

Lemma 3: The outcome of the LCP-replication algorithm is the only static outcome of the caching scheme.

Proof: We first show that no node other than the nodes of the form $0...0a_{p-1}a_{p-2}...a_0$ may cache the file. Suppose node s is not of that form, but caches a copy of the file. It must have $l(s,t) \leq e-p-1$. By the same reasoning as in lemma 1, the fraction of queries for 0 that passes through s must be no greater than $\frac{1}{2^{p+1}}$. Hence, the total rate of query served by s cannot be greater than $\frac{n\lambda_f}{2^{p+1}} = \frac{\theta_f}{2}$. However, by the algorithm, node s should remove the cached copy in this case, which is a contradiction.

Next, consider nodes of the form $0...01a_{p-2}...a_0$. The total rate of queries that reach one such node is precisely θ_f because there are no other upstream nodes on the query paths that cache the file. Hence, each such node must cache the file.

Next, consider nodes of the form $0...001a_{p-3}...a_0$. The total rate of queries that potentially can reach one such node is precisely $2\theta_f$, out of which, θ_f are intercepted by the corresponding node $0...011a_{p-3}...a_0$. No other upstream nodes cache the file. Hence, node $0...001a_{p-3}...a_0$ receives queries at rate θ_f, hence, it caches and serves the file. This argument can be applied inductively, and hence, we conclude that every node of the form $0...0a_{p-1}a_{p-2}...a_0$ receives queries at rate θ_f, hence, it caches and serves the file. ■

Lemma 3 says, if the caching algorithm ever reaches an equilibrium, it must be the outcome of the LCP-replication algorithm. The same argument also shows the equilibrium is actually stable in the sense of the following lemma.

Lemma 4: Given any initial set of nodes that cache the file, the algorithm eventually converges to the LCP-replication outcome.

Proof: Nodes not of the form $0...0a_{p-1}a_{p-2}...a_0$ first remove their cached copies of the file, if they initially have any, due to insufficient queries that pass through them. Then, nodes of the form $0...01a_{p-2}...a_0$ must cache the file for the same reason as in the proof of Lemma 3. Next, nodes of the form $0...001a_{p-3}...a_0$ must cache the file, again for the same reason as in the proof of Lemma 3. Inductively, all nodes of the form $0...0a_{p-1}a_{p-2}...a_0$ must cache the file and each handles queries at rate θ_f. ■

We see that the above caching algorithm is in fact robust. The above results do not require $n\lambda_f/\theta_f$ is an integer power of 2. Suppose $2^p < n\lambda_f/\theta_f < 2^{p+1}$, for some integer $0 \leq p \leq e-1$. Then, the result of dynamic caching algorithm is that the file is cached at the 2^{p+1} nodes of the form $0...0a_pa_{p-1}...a_0$ and each of them handles queries at rate $n\lambda_f/2^{p+1}$, which is less than θ_f but greater than $\frac{1}{2}\theta_f$. The proofs for Lemma 3 and Lemma 4 in this case are essentially the same.

A. Measuring the Rate for Multiple File Requests

Typically, any of the nodes in the peer-to-peer network will see/route requests for multiple files. To be able to apply the load balancing algorithm we we introduced earlier in this section, at least in principle the rate of each file has to be determined and compared with θ_f. Since the number of files stored in the network is usually comparable with the number of nodes, this suggests that each node needs memory proportional to the number of nodes just to keep track of the rates to be able to apply the load-balancing algorithm. This is clearly a scalability problem that can hamper the applicability of our algorithm.

To address the linear dependency of the load-balancing algorithm in the number of nodes, we observe that each node needs only to answer questions of the form: $r(f) > \frac{1}{2}\theta_f + \epsilon$ and $r(f) < \frac{1}{2}\theta_f - \epsilon$. Moreover, in practice it is enough if these questions can be answered with high probability instead of exactly, as long as the probability increases fast as the value of $r(f)$ is away from the boundary. For nodes s for which $l(s,f)$ is large the rate of requests for files f is very small (according to Lemma 1 the rate decreases exponentially with $l(s,f)$). This means that most of the rates for any given node are small; only few are comparable to θ_f. In any unit of time, the problem of tracking the rates larger than a threshold, say $\frac{1}{2}\theta_f - \epsilon$, it is equivalent to the problem of tracking the elements in a string that have frequency higher than a given threshold. These types of queries are called *iceberg queries* in the database literature

[1]. Iceberg queries are closely related to *top-k* queries, i.e. find the k elements with highest frequency, since by setting k to be the ratio of the total number of elements and iceberg threshold, all elements that have frequencies over the threshold will be determined. A number of algorithms for the computation of top-k elements and their frequencies over streaming data using small space have been proposed,[3], [2]. Any of them can be used here to determine the files for which the rate of requests is larger or smaller than the prescribed thresholds using small amounts of space.

V. The Complication with the Number of Nodes

In the last two sections, we have made two assumptions (i) the number of nodes $n = 2^e$ and (ii) every position in the name space is occupied by a node. In this section, we will show that the first assumption is not a serious limitation. However, the second assumption is a more serious limitation, which we will deal with in Section VI.

Without assumption (i), let us write $2^{e-1} < n < 2^e$ for some integer $e \geq 1$. we will show that Lemma 2 is not fundamentally altered. That is, the loads to each cache node are nearly the same under uniform request pattern. We will do so by showing that the properties of the prefix tree are not fundamentally altered. Suppose the query destination is node 0, which is the only node that stores file 0 at the moment. To simplify the analysis in this case, we will not look at the probabilities that a query from a random node passes through each node on its way to the root node 0. Instead, let us assume every node makes one request for 0 and we will count the number of queries seen by different nodes. For node s, let $N(s)$ denote the number of queries seen by node s. Consider a node s at level i on the prefix tree and its level $i+1$ child $\sigma(s)$, for $0 \leq i < e$. In other words, s has the form $0...01a_{i-2}...a_0$, $\sigma(s)$ has the form $0...011a_{i-2}...a_0$. We claim,

Lemma 5:

$$N(s) - 2N(\sigma(s)) \leq 1 \tag{1}$$

Proof: Note that $N(s)$ is the number of queries originating from nodes of the form $*...*1a_{i-2}...a_0$, $N(\sigma(s))$ is the number of queries originating from nodes of the form $*...*11a_{i-2}...a_0$, and $N(s) - N(\sigma(s))$ is the number of queries originating from nodes of the form $*...*01a_{i-2}...a_0$. Suppose the largest node (in terms of node ID) of the form $*...*11a_{i-2}...a_0$ is $b_{e-1}...b_{i+1}11a_{i-2}...a_0$. In other words, $b_{e-1}...b_{i+1}11a_{i-2}...a_0 + 2^{i+1} \geq n$. We claim that $b_{e-1}...b_{i+1}01a_{i-2}...a_0 + 2^{i+2} \geq n$, because

$$\begin{aligned} & b_{e-1}...b_{i+1}01a_{i-2}...a_0 + 2^{i+2} \\ = \; & b_{e-1}...b_{i+1}01a_{i-2}...a_0 + 2^i + 2^{i+1} + 2^i \\ = \; & b_{e-1}...b_{i+1}11a_{i-2}...a_0 + 2^{i+1} + 2^i \\ \geq \; & n \end{aligned}$$

This shows that $N(s) - N(\sigma(s))$ can be greater than $N(\sigma(s))$ by no more than 1. ∎

Lemma 5 implies the number of queries that reach $\sigma(s)$ is nearly 1/2 of the number of queries that reach s. With a similar argument, we can show that the number of queries seen by the same-level nodes may differ by no more than 1.

Lemma 6: For two nodes, say u and v, at the same level, $|N(u) - N(v)| \leq 1$.

Proof: Suppose u and v are at level i, for some $i \in \{2, 3, ..., e\}$. They are of the form $0...01a_{i-2}...a_0$. The largest possible difference between $N(u)$ and $N(v)$ happens when $u = 0...01...1$ and $v = 0...010...0$. It suffices to show $N(v) - N(u) \leq 1$. Let $b_{e-1}...b_i1...1$ be the largest among all nodes whose queries pass through u. Then, $b_{e-1}...b_i1...1 + 2^i \geq n$. Then, it must be true that $b_{e-1}...b_i10...0 + 2^{i+1} \geq n$, because

$$\begin{aligned} & b_{e-1}...b_i10...0 + 2^{i+1} \\ = \; & b_{e-1}...b_i10...0 + 2^{i-1} - 1 + 1 + 2^{i-1} + 2^i \\ = \; & b_{e-1}...b_i1...1 + 2^i + 1 + 2^i \\ \geq \; & n \end{aligned}$$

∎

We will re-state Lemma 2 with a slightly different format.

Lemma 7: Suppose each node in the network makes one query for file t. Suppose at each level, the number queries of seen by a node s is twice as many as the queries seen by node $\sigma(s)$. Suppose the number of queries seen by nodes at the same level is the same. Then, after p replication steps, $p = 1, 2, ..., e$, the number of queries served by each node that contains a copy of the file is identical.

Proof: Without the loss of generality, let us assume the destination is $t = 0$. After the first replication, node 0...01 has a copy of the file. It will intercept exactly half of the queries destined for node 0. Hence, node 0 and node 0...01 serve equal number of queries. Now, suppose the lemma is true for p. The files are copied to all nodes of the form $0...0*...*$, where the last p digits are wild cards. At the $(p+1)^{th}$ step, the file is replicated to nodes of the form $0...01*...*$, where the first 1 from the left occurs at the $(p+1)^{th}$ position from the right. These new nodes are at level-$(p+1)$ of the prefix tree, each of which is connected to a distinct node at a previous level. Hence, half of them have level-p parent nodes. Suppose node v has a level-p parent u. By the assumption of the lemma, $N(v) = N(u)/2$. Hence, right after the $(p+1)^{th}$ replication step, node v serves half of the queries that were served by node u in the p^{th} step. For a node y at level $p+1$ that has a parent w at a level before p, $N(y) = N(v) = N(u)/2 = N(w)/2$, where the first equality is by the assumption of the lemma and the last equality is by the induction hypothesis. We see that, right after the $(p+1)^{th}$ replication step, each of the level-$(p+1)$ new node takes over half of the queries of a node at some previous level. Hence, all cache nodes right after the $(p+1)^{th}$ replication step handle exactly the same number of queries, which is half of the queries handled by each cache node in the previous step. ∎

When the total number of nodes is much larger than the number of the cache nodes, we can ignore the difference of 1 query in Lemma 5 and Lemma 6. The assumptions of Lemma 7 are satisfied (approximately).

VI. LCP-Replication in Randomized Pastry Network

We stress that the desirable load balancing properties of the particular Pastry network we have been discussing depend

crucially on the particular choice of routing specified in Section II-B. To illustrate this, suppose we choose the largest node (in term of node ID) among all eligible nodes for each routing table entry in Table I. Then, the two next-hop nodes in level i are $a_{e-1}...a_{e-i+1}01...1$ and $a_{e-1}...a_{e-i+1}11...1$. The result is that, no matter where the query for 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 $01...1 \rightarrow 001...1 \rightarrow ... \rightarrow 0...01 \rightarrow 0...0$. Hence, there is no load-balancing at all in this network regardless where the file is replicated, unless every node contains a copy of the file.

The example suggests that, to achieve ideal load balancing, one must be very careful in constructing the routing table. Our routing table shown in Table II requires that every position in the name space has a node. In typical file-sharing applications, the nodes only thinly populate the name space, and hence, many required nodes for the routing table most likely do not exist. We will handle the situation by constructing a random network, which will have the desired load balancing properties as the deterministic network, but only when averaged over all random instances of the network. In other words, we will choose eligible neighbors at random, resulting in randomized routing tables.

A. Randomized Routing Tables

Starting with the generic Pastry 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, the next-hop node at level 2 for digit value 0 is chosen uniformly at random from all *available* nodes of the form $a_{e-1}0*...*$. Note that the resulting network is an instance of a random network. After the network is constructed, the routing rule is still as specified in Section II-B. We will show that, regardless the name space is fully populated by nodes or not, the random network has the desirable load balancing properties similar to the previous deterministic network, whose routing table is given by Table II. Consider queries for file 0.

Lemma 8: Suppose the file is replicated at all nodes $0...0*...*$ with p wild cards, $p = 0, 1, ..., e$. For any query originating from a node that is not one of the cache nodes above, it is equally likely to be served by any of the cache nodes.

Proof: First, consider a query from a node of the form $0...01a_{p-1}...a_0$. At the first hop of the routing, it will be routed to one of the cache nodes with equal probability. Next, suppose the lemma is true for queries originated from any node of the form $0...01*...*$, with the leading 1 at the i^{th} position from the right, where $p+1 \leq i \leq q < e$. Consider nodes of the form $0...01*...*$ with the leading 1 at the $(q+1)^{th}$ position. After the first hop, the query is routed to a node of the form $0...0a_{q-1}...a_0$ with equal probability. Either this node is one of the cache node, or it isn't. If it isn't, then it is as if the query starts from that node, and by the induction assumption, will be served by one of the cache nodes with equal probability. Hence, the original query will be served by one of the cache nodes with equal probability. ■

An easy corollary of the lemma is,

Corollary 9: Suppose the file is replicated at all nodes $0...0*...*$ with p wild cards, $p = 0, 1, ..., e$. 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.

B. Simulation Experiments

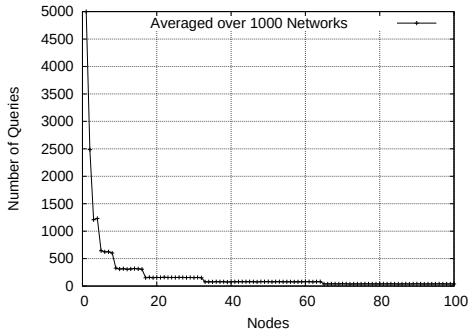

Fig. 2. Average number of queries seen by each node in a random network, averaged over 1000 instances. The network has 5000 nodes. The name space size is also 5000

1) Case 1: Name Space Size $= 5000$, $n = 5000$, *No Caching:* We now show simulation experiments to show the effectiveness of the LCP-replication strategy on the random network. In the first case, we have a network with 5000 nodes, fully populating the name space of size 5000. Figure 2 shows the query count at each node averaged over 1000 instances of a random network. In each instance, only the root node contains the file of interest and each node generates precisely one query for the file. We see that the load to each node decreases by half for nodes in group of 2^p, $p = 0, 1, ..., e-1$, which is the expected behavior. As a comparison, Figure 3 shows the query count at each node in two different instances of the random network. Even though the general trend for the query counts still decreases exponentially in each instance, the precise query counts do have some fluctuation around the expected values.

2) Case 2: Name Space Size $= 5000$, $n = 5000$, *with Caching:* Next, consider the same 5000-node random network, in which the requested file is cached in 32 nodes according to the LCP-replication scheme. In other words, the file is replicated at node 1 to 31. When averaged over instances of the random network, we should expect that node 0 to 31 each serve the same amount of queries. This is indeed verified by Figure 4. Compared to Figure 2, the load to node 0 is reduced by a factor of 32. When we look at the query counts on instances of the random network, as shown in Figure 5, we do get load reduction to node 0. However, the number of queries served by different cache nodes varies more and also depends on the particular network instance.

We know that, when averaged over different network instances, the query count seen by each node agrees with the theoretical results. We would like to investigate its fluctuation. Figure 6 (a) and (b) show the probability mass function of the query counts seen by node 1 and node 63, respectively, collected over 10000 random network instances. It seems that the variances of the query counts are not negligible. We hope that these experimental results will guide us in our future analysis on the probability distribution of the random network.

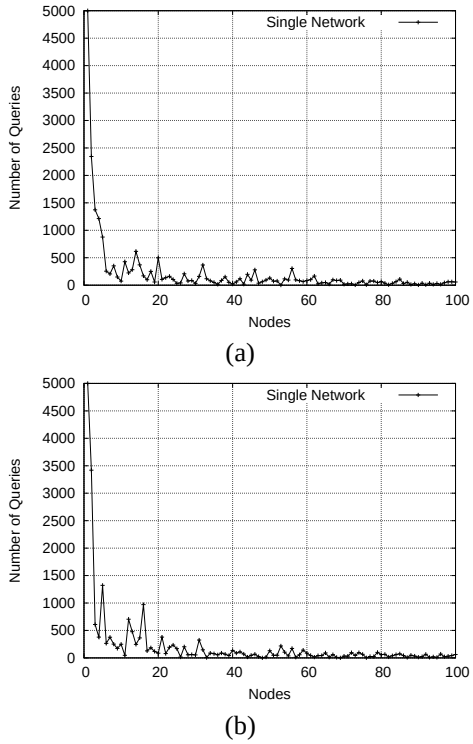

Fig. 3. Number of queries seen by each node in instances of a random network with 5000 nodes. The name space size is also 5000. (a) network 1; (b) network 2

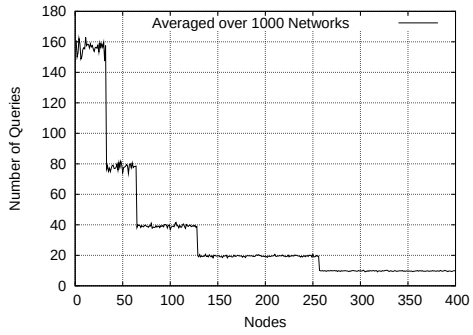

Fig. 4. Average number of queries seen by each node in a random network, averaged over 1000 instances. The file is cached at node 0 to 31. The network has 5000 nodes, and the name space size is 5000.

3) Case 3: Name Space Size $= 5000$*,* $n = 500$*, No Caching:* We next move to the case where 500 nodes sparsely populate a name space of size 5000. Recall that this type of situations create problems for the routing shown in Table II. The nodes are distributed uniformly in the name space. Figure 7 shows the query count seen by each node averaged over 1000 network instances. We still see the exponential decrease in the query counts for groups of nodes whose numbers increase exponentially. Note that, unlike the previous two cases, the horizontal axis does not correspond to the node ID. It shows the indices of the nodes in increasing order of their ID's. Figure 8 shows the query count for a fixed network instance.

4) Case 4: Name Space Size $= 5000$*,* $n = 500$*, with Caching:* Now, we cache the file at nodes whose ID's are less

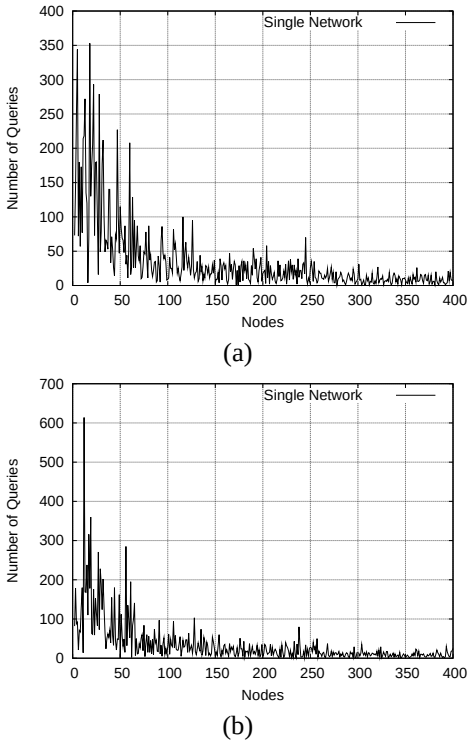

Fig. 5. Number of queries seen by each node in instances of a random network. The file is cached at node 0 to 31. The network has 5000 nodes, and the name space size is 5000. (a) network 1; (b) node 2

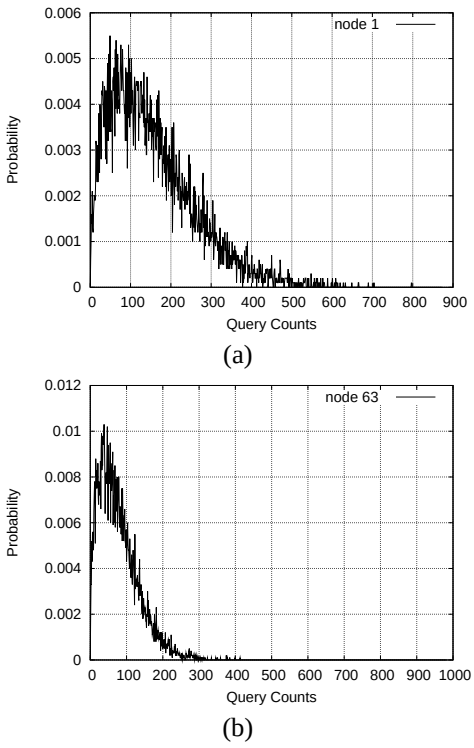

Fig. 6. Probability mass function of the number of the query counts. The file is cached at node 0 to 31. The network has 5000 nodes, and the name space size is 5000. (a) node 1; (b) node 63

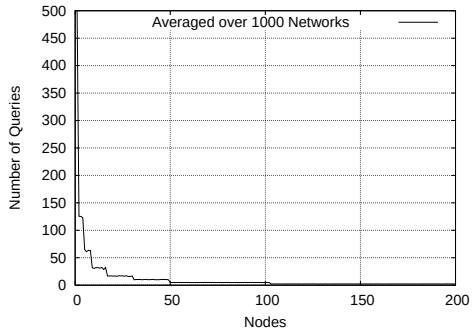

Fig. 7. Average number of queries seen by each node in a random network, averaged over 1000 instances. The network has 500 nodes. The name space size is 5000.

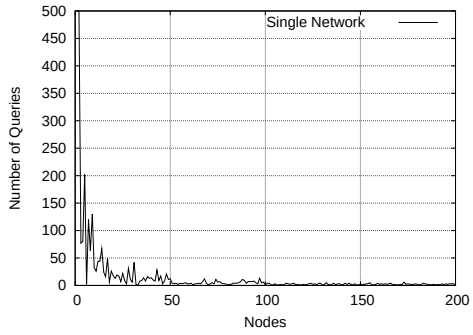

Fig. 8. Number of queries seen by each node in a fixed instance of a random network. The network has 500 nodes. The name space size is 5000.

than 128. The number of such nodes is a random variable, which is a function of the network instance. Figure 9 and Figure 10 demonstrate that LCP-replication achieves nearly perfect load balancing in the average sense, and achieves acceptable load balancing in fixed instances of the network.

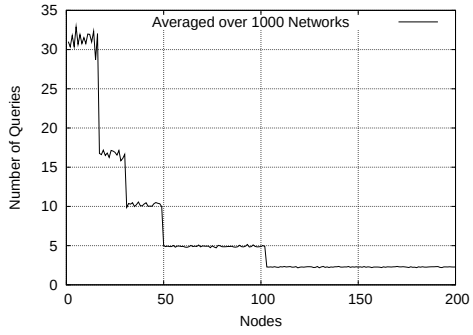

Fig. 9. Average number of queries seen by each node in a random network, averaged over 1000 instances. The network has 500 nodes. The name space size is 5000. The file is cached at nodes whose ID's are less than 128.

VII. Conclusions

We have specified two routing schemes in pastry-type networks, which allow us to design a provably ideal load balancing scheme, LCP-replication. That is, the load to each cache node is perfectly balanced under certain qualifying

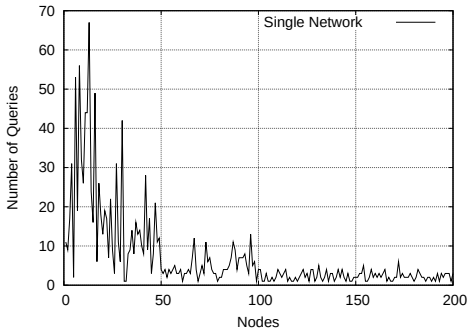

Fig. 10. Number of queries seen by each node in an instance of a random network. The file is cached at node 0 to 31. The network has 5000 nodes, and the name space size is 5000.

conditions. We have also extended the LCP-replication scheme to a more automatic and more dynamic caching scheme. In essence, each node decides to cache a copy of the file if its load exceeds a threshold. This should not be construed as a naive caching scheme, because its success depends crucially on the routing tables we set up. Preliminary experimental results on the automatic caching scheme appear to be very promising, which we will pursue further. The statistical properties about the random network in Section VI requires further attention. Besides practical implication on P2P file-sharing applications, our work should also deepen the understanding of Pastry-like network. Possible extension of our work to Chord-type or CAN-type networks should also be studied.

References

[1] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 299–310, 24–27 1998.

[2] Cheqing Jin, Weining Qian, Chaofeng Sha, Jeffrey X. Yu, and Aoying Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 287–294, 2003.

[3] G. Manku and R. Motwani. Approximate frequency counts over data streams, 2002.

[4] C. Plaxton, R. Rajaraman, and A Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA*, pages 311–320, Newport, Rhod Island, June 1997.

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

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

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

[8] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, University of California University, Berkeley, Computer Science Division (EECS), April 2001.