Probabilistic Location and Routing

Sean C. Rhea, John Kubiatowicz

***Abstract*—We propose *probabilistic* location to enhance the performance of existing peer-to-peer location mechanisms in the case where a replica for the queried data item exists close to the query source. We introduce the *attenuated Bloom filter*, a lossy distributed index. We describe how to use these data structures for document location and how to maintain them despite document motion. We include a detailed performance study which indicates that our algorithm performs as desired, both finding closer replicas and finding them faster than deterministic algorithms alone.**

I. Introduction

Today's exponential growth in network bandwidth and storage capacity has inspired a whole new class of distributed, peer-to-peer storage infrastructures. Systems such as Farsite [1], Freenet [2], Intermemory [3], OceanStore [4], CFS [5], and PAST [6] seek to capitalize on the rapid growth of resources to provide inexpensive, highly-available storage without centralized servers. The designers of these systems propose to achieve high availability and long-term durability in the face of individual component failures through replication and coding techniques.

Although wide-scale replication has the potential to increase availability and durability, it introduces two important challenges to system architects. First, if replicas may be placed anywhere in the system, how should we *locate* them? Second, once we have located one or more replicas, how should we *route* queries to them? We can formulate the combination of these two operations as a single *location and routing* problem that efficiently routes queries from a client to the closest replica adhering to certain properties, such as the replica with the shortest network path to the client or the replica residing on the least loaded server. In many cases, combining location and routing into a single, compound operation yields the greatest flexibility to route queries quickly with minimal network overhead. The importance of such *location-independent routing* techniques is well recognized in the community, and several proposals such as CAN [7], Chord [8], Pastry [9], and Tapestry [10] are currently under study.

These existing schemes share the characteristic that in the worst case, a location and routing operation requires $O(\log N)$ sequential network messages to search a distributed system of N servers. Some of these algorithms use substantially fewer messages to perform their task in the common case. This scalability is commendable, and it allows for the total query routing time to be close to optimal *when the replica is far from the query source*. However, as the replica approaches the location of the query source, the performance of the existing algorithms quickly diverges from optimality. This divergence is easy to understand: a small amount of "mis-routing" in the local area can lead to a large divergence from optimality, since the optimal path is short to begin with.

S. Rhea and J. Kubiatowicz are with the University of California, Berkeley. Email: {srhea, kubitron}@cs.berkeley.edu

Currently, such systems make only meager attempts to place replicas for network locality, and the sizes of the documents they locate are on the order of megabytes, so this poor performance in locating nearby replicas does not significantly affect overall document retrieval time. However, in the OceanStore system [4], we intend store documents whose sizes are as small as a few kilobytes and to go to great lengths to place those documents near their query sources. In such a situation, nearby location performance can be a large component of the overall retrieval time.

In this paper we present a *probabilistic* location and routing algorithm designed to enhance the performance of existing deterministic wide-area location mechanisms. A probabilistic algorithm is one that may fail to discover a replica for a given document even when such replicas exist; for example, it might find "nearby" replicas with high probability, but fail if no replica is nearby. Assuming that the probabilistic algorithm finds replicas quickly when it *can* and fails quickly when it *cannot*, we can enhance the performance of the location and routing process through a hybrid approach: first try the probabilistic algorithm, then follow with the deterministic algorithm if needed.

Our probabilistic location and routing algorithm is based on *attenuated Bloom filters* and has the following properties:

- *It is decentralized.* It requires no central point of control and is thus suitable for use in the peer-to-peer systems for which it is intended.
- *It is locality aware*. If a query site lies close to a replica for the queried document, our algorithm finds that replica with high probability.
- *It follows a minimal search path*. With high probability, our algorithm follows the shortest path between a query site and the replica that satisfies the query.
- *It uses constant storage per server*. The amount of storage used at each server in the system is small and constant in the number of documents indexed.

This later property allows each hop in the query path to proceed without high-latency disk accesses, further enhancing the speed of operation. When used with a deterministic algorithm, attenuated Bloom filters allow us to achieve the "best of both worlds": quickly finding nearby replicas when they exist, yet finding every document even when replicas are scarce.

Figure 1 shows the potential of our technique. This graph illustrates the *stretch* of two possible algorithms: a real deterministic algorithm (solid line) and a hybrid combination of attenuated Bloom filters with the same deterministic algorithm (dotted line). In this context, *stretch* is a measure of the overhead of location-independent routing: it is the ratio of actual time to route a query through the infrastructure versus the ideal network latency (on the underlying IP network) to the closest replica. This graph will be further discussed later, but the basic

0-7803-7476-2/02/$17.00 (c) 2002 IEEE.

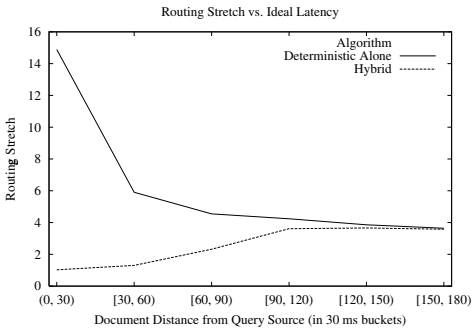


Fig. 1. *The Big Picture.* Stretch as a function of distance between query source and replica. *Stretch* measures the overhead of combined routing and location relative to ideal (network) latency–hence, lower is better. At short distances, the routing stretch of the deterministic algorithm (solid line) is greatly reduced when coupled with a probabilistic algorithm (dotted line).

message is simple: attenuated Bloom filters reduce latency for the short-distance case, effectively smoothing out the overall response time.

This paper makes the following contributions. First, we introduce *attenuated Bloom filters*, data structures that reside at each node in the system. We present the *query* algorithm, which routes queries from node to node in search of a replica, and the *update* algorithm, which propagates location information from filter to filter. Second, we couple attenuated Bloom filters with two different deterministic wide-area location algorithms. We use a detailed performance simulation to explore the behavior of this hybrid approach both on a random, static arrangement of replicas and on a dynamically changing allocation of replicas modeling traffic to Web caches. We show both that our probabilistic algorithm finds closer replicas and that it finds them faster than a deterministic algorithm alone. Furthermore, we show that the additional bandwidth required by the probabilistic algorithm is reasonable.

The remainder of this paper is as follows. Section II describes attenuated Bloom filters in detail. Section III presents our simulation environment, and Section IV describes our experimental results. Section VI describes related work. Section V postulates some future work, and Section VII concludes.

II. Algorithm Description

In this section we present our probabilistic location algorithm. This algorithm works via an overlay network between participating servers. Each server has a set of *neighbors*, chosen from the participating servers closest to it in network latency. A server associates with each neighbor a probability of finding each document in the system through that neighbor. This association is maintained in constant space using a data structure we call an *attenuated Bloom filter*. The set of these probabilities forms a potential function over the servers in the system; location is a simple matter of climbing this function to a server with the desired data item.

In the following, we briefly summarize Bloom filters. We continue by introducing *attenuated Bloom filters*, describing their use for replica location, then finish with algorithms for updating filters as replicas move.

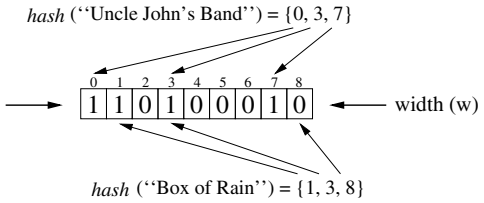


Fig. 2. *A Bloom Filter.* An array of w bits that serve to summarize a set of objects. To check an object's name against a Bloom filter summary, the name is hashed with n different hash functions (here, $n = 3$) and the bits corresponding to the result are checked in the filter. In this picture, the represented set probably contains the name "Uncle John's Band", since bits 0, 3, and 7 are all true. It definitely does not contain "Box of Rain", however, since bit 8 is false.

A. Bloom Filters

Bloom filters are an efficient, lossy way of describing sets [11]. A Bloom filter is a bit-vector of length w with a family of independent hash functions, each of which maps from elements of the represented set to an integer in $[0, w)$. To form a representation of a set, each set element is hashed, and the bits in the vector associated with the hash functions' results are set. To determine whether the set represented by a Bloom filter contains a given element, that element is hashed and the corresponding bits in the filter are examined. If any of the bits are not set, the represented set definitely does not contain the object. If all of the bits are set, the set *may* contain the object; there is a non-zero probability that it does not, however. This case is called a *false positive*, and the false positive rate of a Bloom filter is a well-defined, linear function of its width, the number of hash functions and the cardinality of the represented set. Figure 2 shows a sample Bloom filter.

If the cardinality of the represented set is a significant fraction of the width w, then the Bloom filter becomes *overloaded*: the rate of false positives increases to the point that the filter is essentially useless. Several studies, such as [12], have explored this phenomenon. In particular, the point at which approximately half of the bits are set is an optimal tradeoff between filter storage and accuracy; however, wider filters are always more accurate.

B. Attenuated Bloom Filters

An *attenuated Bloom filter* of depth d is an array of d normal Bloom filters. As mentioned earlier, we assume that each node in the system has a set of overlay neighbors participating in the probabilistic location algorithm. In the context of our algorithm, we associate each neighbor link with an attenuated Bloom filter. The first filter in the array summarizes documents available from that neighbor (one hop along the link). The ith Bloom filter is the merger of all Bloom filters for all of the nodes a distance i through any path starting with that neighbor link, where distance is in terms of hops in the overlay network[1]. Figure 3 shows the attenuated Bloom filter that Node A would associate with Node B in the given network. For example, both "Uncle John's Band" and "Sugar Magnolia" are two hops away from Node A through Node B, so the second level of filter F_{AB} contains true values at all bits in the union of those documents' hash values (0, 2, 3, 5, 7).

[1]The astute reader might surmise that all links ending in a particular node have the same attenuated filter associated with them. This is true for now, but will change when we introduce selective updates in Section II-D.

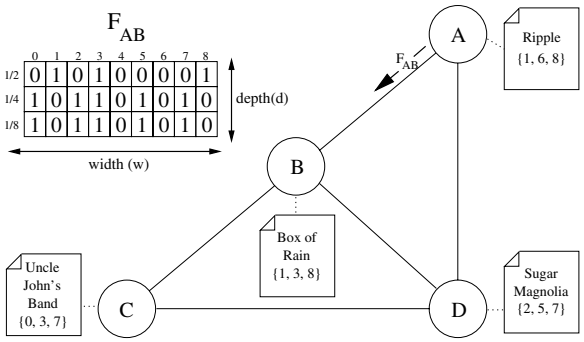


Fig. 3. *Attenuated Bloom Filters.* An attenuated Bloom filter is an array of d Bloom filters, each of width w. Component filters are labeled with their *level* in the array (top filter is level 1). Each outgoing link (say, A $\rightarrow$ B) has an attenuated filter associated with it (F_{AB}). Level 1 summarizes replicas on the neighbor at the end of the link. Level 2 summarizes replicas that are two-hops away along that link, etc. We assign a potential value to each level (here $\frac{1}{2}$, $\frac{1}{4}$, ...). Higher levels are thus *attenuated* with respect to lower levels.

To map from an attenuated Bloom filter to a potential value, one queries each level for a document's name. The levels are assigned geometrically decreasing potential values; the value of the potential function of a filter for a given document is the sum of all of the potential values for the levels of the filter which contain the document. For example, in F_{AB}, the document "Uncle John's Band" would map to the potential value $1/4 + 1/8 = 3/8$, since it is reachable through Node B in two and three hops. We say that higher filter levels are *attenuated* with respect to earlier filter levels, hence the name "attenuated Bloom filter". We refer to filters with only one level as *nonattenuated*.

C. The Query Algorithm

As mentioned above, we associate an attenuated Bloom filter with each outgoing neighbor link. To perform a location query, the querying node examines the 1st level of each of its neighbors' attenuated Bloom filters. If one of the filters matches, it is likely that the desired data item is only one hop away, and the query is forwarded to the matching neighbor closest to the current node in network latency. If no filter matches, the querying node looks for a match in the 2nd level of every filter. As before, if a match is found, the query is forwarded to the matching neighbor of lowest latency. This time, however, it is not the immediate neighbor who is likely to possess the data item, but one of its neighbors. This next neighbor is determined as before, by examining the attenuated Bloom filters of the current server.

A filter of depth d by definition stores information only about servers d hops from the current server. For this reason, if a query were to reach a server d hops from its source due to a false positive, there is no incentive to forward it further. In other words, since the query reached the particular server that it did through error, any further information about which nearby servers might contain the desired data item may likely be incorrect as well. When such circumstances arise, there remain two possibilities for finding the data item. The probabilistic algorithm can simply give up and forward the request to the deterministic algorithm, or the query can be returned to the previous server in the query path to be sent on to the next best neighbor. In this way, the query algorithm comes to resemble a depth-first search which is guided by the potential function represented by the attenuated Bloom filters.

Since the purpose of the probabilistic algorithm is to improve latency in the case where nearby replicas exist, we view this latter solution as overreaching, since the time required for it in the worst case is possibly slower than the deterministic algorithm, with less certain results. As a result, we do not allow backtracking, and after d unsuccessful hops we immediately defer to the deterministic algorithm.

Finally, in situations where d is large, a false positive may cause a query to return to a server it has already visited. For this reason, each query in the system contains a list of all of the servers that it has visited so far, and servers never forward a query to a server it has already visited. Since we do not allow backtracking, this list is at most d elements long, so the cost of this optimization is small.

D. The Update Algorithm

For the query algorithm to be successful, the attenuated Bloom filters at each node that direct queries must be kept up-to-date. Every time a new data item is added to a server, there is a possibility that the Bloom filter representing the set of data items it stores will change as well. If such a change occurs, neighbors of the server will only find the new data item if the change is propagated to them in some manner. The way in which this change is propagated is the *update algorithm*.

The fundamental observation behind the update algorithm is that unless the Bloom filters are loaded to a degree such that they are no longer useful for location, an update to a single server in a system with filters of depth d should eventually change at least one bit in the filters of every server within d hops of the update site. Ideally, this propagation is a single wave spreading from the source of the change outward. Moreover, updates due to different data items involve only a small number of common bits. Thus there is little benefit to combining updates, except to save on the network costs associated with sending many small messages instead of one large one. We therefore assume that all updates occur independently, except for the possibility that updates destined for the same server may be grouped into the same network message.

An update proceeds as follows. Every server in the system stores both an attenuated Bloom filter for each outgoing link (*e.g.* F_{AB} in Figure 3), and a copy of its neighbor's view of the reverse direction. When a new document is stored, the server calculates the changed bits in its own filter and in each of the filters its neighbors maintain of it. It then sends these bits out to each neighbor; this is a form of *diff compression*. On receiving such a message, each neighbor attenuates the bits one level and computes the changes they will make in each of its own neighbors' filters. These changes are then sent out as well. One can thus view an update as the set of changed bits propagating outward from the source of the change.

One problem with this algorithm as currently specified is that unless the overlay network is a tree rooted at the update source, the update will be propagated to some servers more than once, wasting network bandwidth and placing redundant information in the filters of the receiving node. For example, consider Figure 3. If a document were added to Node D, Node A would receive the corresponding update three times: first from D directly, then through B, and again through B via C. This redundancy would place unnecessary load on the network. Moreover,

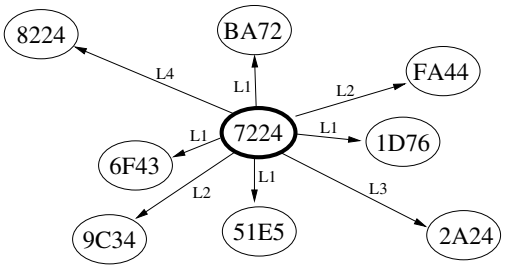


Fig. 4. *Tapestry Routing Mesh.* Each node is linked to other nodes via *neighbor links*, shown as solid arrows with labels. Labels denote which digit is resolved during link traversal. Here, node `7224` has an L1 link to `BA72`, resolving the first digit, an L2 link to `FA44`, resolving the second digit, etc.

it would needlessly place information about the new document in all three levels of F_{AB}. The lower levels of an attenuated Bloom filter represent the combined documents of many more nodes than the higher-potential ones, so this redundancy of information is quite detrimental to the false positive rate.

We can thus imagine filtering updates to improve both the bandwidth utilization and the load on lower filter levels. This filtering changes the information stored in the attenuated filters, thus altering the semantics of these filters slightly. As we will show in Section IV, however, we can continue to use the query algorithm of Section II-C and achieve lower update bandwidth.

We describe two distinct update filtering algorithms; we call the naive approach already described the *no filtering* case. To filter, we tag all updates with an identifier consisting of their source node and a monotonically increasing sequence number. We then perform the following types of filtering:

- *destination filtering:* Destination servers remember the identifiers of every update they see for a short period, allowing them to ignore subsequent arrivals of an update through different paths. This filtering prevents redundant information in the destination's neighbor filters.
- *source filtering:* Once a server receives a duplicate update from one of its neighbors, it sends a message to that neighbor to inform it of this redundancy. The neighbor stops forwarding new updates from that same source.

Both of these techniques save network bandwidth. The second, however, is somewhat more sophisticated, since it performs a form of topology discovery, squelching update messages before they are even sent. The additional information stored for source and destination filtering is soft state and is periodically flushed so as to adapt to changes in the overlay network.

As a final point, we note that update filtering introduces a bit of subtlety with respect to replica deletions. When a replica deletion causes bits at any level of a Bloom filter to transform from one to zero, we must be careful to propagate this deletion to all appropriate nodes. This may, in some cases, involve ignoring update filters that have been previous installed. Figure 16 in the Appendix describes the complete algorithm.

III. Experimental Setup

To test the effectiveness and cost of our probabilistic algorithm, we simulated it in conjunction with two different deterministic algorithms for location-independent routing. In this section we describe the deterministic algorithms, then discuss our simulation environment and experiments. The results of our simulation are provided in Section IV.

A. Deterministic Location and Routing Algorithms

We used two different deterministic algorithms to provide our probabilistic algorithm with the greatest variety of "competition". The first is *home-node location*, an idealized architecture that resembles a combination of DNS [13] and optimized directory-based cache coherence [14]. The second is Tapestry [10], an actual distributed, wide-area location and routing infrastructure with interesting locality properties.

1) Home-Node Location Overview: Our first deterministic algorithm postulates that every document in the system has a *home-node server* that keeps a set of pointers to every replica of the document. To route a query to a replica, a client sends the query to its home node, which forwards the query to the replica closest to the client.

We chose this architecture for two reasons. First, it is a very simple; as we will see in the next section, more realistic architectures are far more complicated. Second, it is an idealized form of directory service (such as DNS), but with oracle-level knowledge of the network topology. In contrast to existing, nonidealized protocols which route to $O(\log n)$ randomly distributed locations in the network to reach a document, this idealized algorithm uses only $O(1)$ such hops, and provides better replica placement relative to the query source.

Of course, several aspects of this architecture are idealized. We do not address the type of infrastructure that a client would utilize for finding the home-node server; we assume that some form of document-to-home-node mapping service is available. Further, since this is a "best-case architecture", we do not address how the directory server keeps its information about replicas current, or how it is able to select the replica closest to the query source. The next section details a distributed directory technique that does not require these idealizations.

2) Tapestry Overview: The wide-area location and routing infrastructure of OceanStore is Tapestry [10], an IP overlay network with a distributed, fault-tolerant architecture. With Tapestry, a query is routed from node to node until the location of a replica is discovered, at which point the query proceeds to that replica. Tapestry differs from the home-node architecture in two distinct ways: (1) Tapestry distributes the directory lookup process in a document-specific way. This removes the need for a separate document-to-home-node mapping. (2) Once Tapestry has discovered the location of a replica, it forwards the query to the replica closest to the point of discovery, rather than to the replica closest to the original query source. This optimizes one-way latency from the query to the replica, but may not optimize subsequent traffic from replica back to query source.

Tapestry begins with the assumption that every server and document in the system can be named with a unique, location-independent identifier, represented as a sequence of hexidecimal digits. We will refer to *node-IDs* for the node names and *globally unique identifiers* (GUIDs) for the documents. Among other things, this means that every query has a unique destination GUID. Tapestry has two major components: a *routing mesh* and a *distributed directory service*.

The Tapestry *routing mesh* is an overlay network between participating nodes. Figure 4 shows a portion of this mesh. Every Tapestry node is connected to other Tapestry nodes via *neighbor links* of various levels. Level-1 edges from a given node connect to the 15 nodes closest (in network latency) with

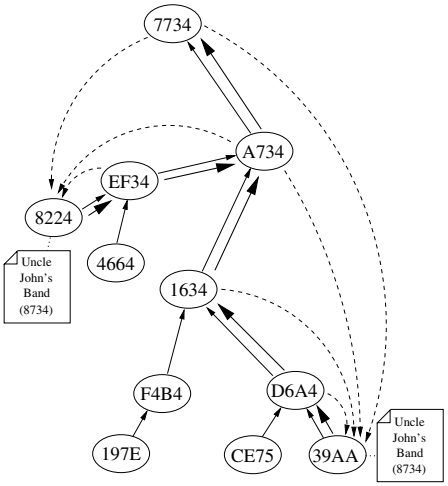


Fig. 5. *Publication in Tapestry.* To publish document `8734`, server `39AA` sends publication request towards the root, leaving a pointer to itself at each hop. Server `8224` publishes its replica similarly.

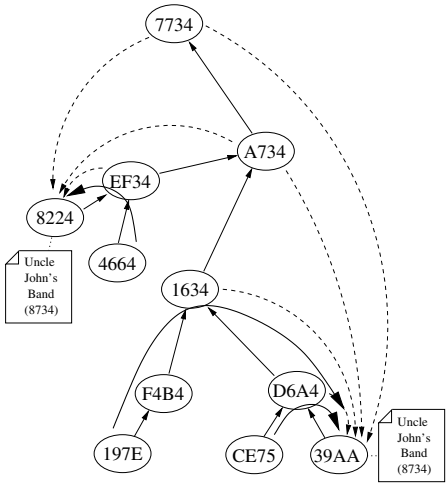


Fig. 6. *Location in Tapestry:* Three different location requests. For instance, to locate GUID `8734`, query source `197E` routes towards the root, checking for a pointer at each step. At node `1634`, it encounters a pointer to server `1634`.

different values in the lowest digit of their addresses. Level-2 edges connect to the 15 closest nodes that match in the lowest digit and have different second digits, *etc.*

Tapestry neighbor links provide a route from every node to every other node in the system: simply resolve the destination node address one digit at a time, using a level-1 edge for the first digit, a level-2 edge for the second, and so forth. This routing scheme is based on the hashed-suffix routing structure presented by Plaxton, Rajaraman, and Richa [15]. While the Tapestry infrastructure includes algorithms for building this neighbor graph dynamically, we assume in this work that the graph is built at the beginning of our simulation and does not change.

To perform location-independent routing, Tapestry deterministically maps each document GUID to a set of unique *root* nodes[2]. In this paper we assume a single root node for each GUID. Thus every unique document and query for that document is associated with a single root node-ID. We use the routing mesh described above to reach the root from any other node in the system; this routing process defines a unique *location tree* for every choice of root node.

Storage servers *publish* the fact that they are storing a replica by *routing* a message toward the root node, depositing *location pointers* to the object at each hop. Figure 5 illustrates two replicas with the same GUID (`8734`) exported by server nodes `8224` and `39AA`. Location pointers are shown as dotted arrows back to servers. Note that both the root note (`7734`) and node `A734` have knowledge of both replicas.

As shown in Figure 6, queries route toward the root node until they encounter a location pointer, then route to the located replica. If multiple pointers are encountered, the query proceeds to the closest replica. The figure shows three different location paths. In the worst case, a location operation involves routing all the way to the root. However, if the desired object is close to the client, then the query path will intersect the publishing path before reaching the root with high probability. In fact, it is shown in [15] that the average distance traveled in locating an object is *proportional* to the distance from that object[3].

[2]Since the node-ID space is sparse, this cannot be a one-to-one mapping. Suffice it to say that there is a way to map GUIDs to root node-IDs, even with dynamic insertion and removal; see [10].

B. Simulation Environment

Our simulator models the physical network as a graph, each edge of which has two values associated with it, α_{net} and β_{net}. To send a message along an edge takes $\alpha_{net} + s\beta_{net}$ seconds, where s is the size of the message in bytes. To send a message along a path of more than one hop takes $\alpha'_{net} + s\beta'_{net}$ seconds, where α'_{net} is the sum of the α_{net} values for every edge along the path, and β'_{net} is the largest β_{net} value of any edge along the path. We do not measure queuing effects or computation time at servers.

Using this simulator, we constructed a physical network topology using the *transit-stub* model of GT-ITM [16]. This topology mimics the structure of large networks observed in nature by dividing the graph into two classes of nodes, called transit nodes and stub nodes. An example transit-stub graph is shown in Figure 7. Transit nodes are grouped into highly connected transit domains, and off each transit node several stub domains are connected. These stub domains are collections of stub nodes which are generally more lightly connected than the nodes in the transit domains. In addition to this general layout, there are several inter-stub domain edges in each graph. We augment the GT-ITM model with bandwidth numbers as follows. All stub to stub edges are 100 Mb/s, all stub to transit edges are 1.5 Mb/s, and all transit to transit edges are 45 Mb/s. These values were chosen to model Fast Ethernet, T1, and T3 connections, respectively. In our experiments, we focus on stub to transit domain bandwidth consumption, since these interdomain edges are the most bandwidth constrained in the system (and in most real systems as well).

Our simulations use transit-stub graphs with six transit domains of ten nodes each. Each transit node has seven stub domains of an average of twelve nodes each, yielding a total of 5,100 nodes per graph. The transit domains are fully connected to each other, and each pair of nodes internal to a domain are connected with probability 0.6. Each pair of stub nodes within

[3]Experiments show a small constant of proportionality; See [10].

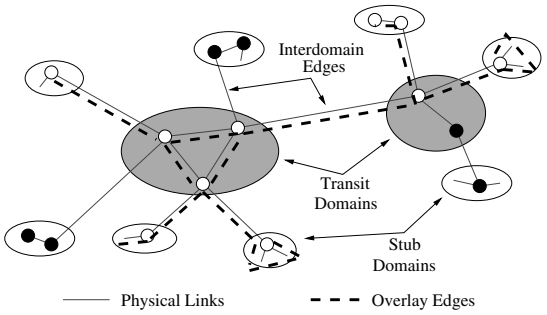


Fig. 7. *A Transit-Stub Graph.* This topology mimics the structure of large networks observed in nature. Shown also is an overlay network which minimizes the number of interdomain edge crossings. Such overlays allow the topology discovery properties of the source filtering algorithm to minimize interdomain bandwidth consumption. See Section IV for details.

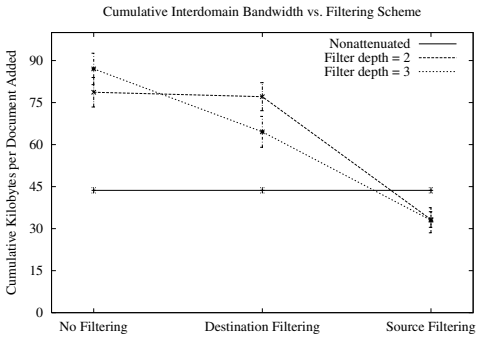


Fig. 8. *Update Bandwidth vs. Update Algorithm.* Update bandwidth includes Tapestry messages and Bloom filter updates. The source filtering algorithm generally uses significantly less bandwidth than no filtering or destination filtering. Neither filtering algorithm effects filters of depth one, since a loop must exist for the algorithms to filter anything. Likewise, destination filtering only helps on loops of three or larger, so it has no effect of filters of depth two.

a stub domain are connected with probability 0.3. We used GT-ITM to generate seven graphs given these parameters to insure that our results were not dependent on the particularities of any one graph.

On top of this physical network, we built Tapestry and probabilistic location overlay networks as follows. We chose 1,000 of the total nodes in the graph uniformly at random without replacement and made them Tapestry servers. We then assigned node-IDs to these servers at random, and built the neighbor graph as described in Section III-A. In some experiments, we also attached probabilistic location servers to these same nodes, using the construction algorithm described in Section II. Finally, in some experiments we further restrict the Bloom filter overlay network to have a minimal number of interdomain edges, while maintaining its average connectivity. The effects of this restriction are described in Section IV.

C. Experiment Descriptions

In this work we document two groups of experiments, called the *static* and the *dynamic* experiments, based on whether the set of replicas in the system changes during the test.

In the static experiments, we randomly place 70 unique files on each of the nodes in the system which are participating in the location protocols. We chose this small number for simulation simplicity; since the optimal size of the Bloom filters scales linearly with the number of documents indexed, our results generalize in a straightforward manner. We allow for all of the location directories to be updated, then arrange for each participating node to request a different set of 12 documents, randomly chosen from the full set. We observe the average location latency versus the minimum possible latency given the constraints of the network. After all of the location requests are complete, we add one new data object to each participating node and observe the network bandwidth used in updating all of the Bloom filters in the system. Each of these experiments is repeated using Bloom filters of several different sizes and depths, using each of the two deterministic algorithms, and using different transit-stub graphs.

To explore the advantages of attenuation, we fix the average number of nodes reachable through the overlay (we present results for 20 reachable nodes) while varying the depth of the filters. Consequently, higher levels of attenuation imply a lower average out-degree in the overlay network (*i.e.* each node has fewer immediate neighbors). Moreover, in many our experiments, we fix the total amount of local storage used by the index; this quantity is the product of the filter width, depth, and number of immediate neighbors per node.

For the dynamic experiments, we used the SURGE web traffic generator [17] to choose our file sizes and reference stream. This generator produces read requests with characteristics similar to observed patterns of web traffic across multiple clients. We used it to produce a trace against 50,000 unique files, with sizes ranging from 75 bytes to 8.69 MB, distributed according to a hybrid of lognormal and pareto distributions (see [17] for more details). The average file size is around 21 kB.

Each node maintains an in-memory cache of the data items it reads, managed in simple least-recently-used (LRU) order. Each cache is 420 kB in size, allowing an average of 20 files to be cached at one time. Additionally, each file in the trace is kept permanently on a simulated hard drive of exactly one node, resulting in each node storing 50 files on its drive, for a total of 70 files between the cache and the disk, to match the static experiments. Data not found in cache is loaded off this drive, which is modeled as a delay of $\alpha_{disk} + s\beta_{disk}$ seconds, where s is the size of the file, $\alpha_{disk} = 10$ milliseconds, and $\beta_{disk} = 100$ nanoseconds per byte. These parameters are similar to those observed on modern drives.

As with the static experiments, the dynamic experiments are performed over a range of Bloom filter sizes and depths and using various different transit-stub graphs and server placements. During these tests, we observe the average time to find a replica and the distance to that replica versus the network distance to the closest replica. We also observe the total interdomain bandwidth consumed.

IV. Results

In this section, we utilize the results of our experiments to justify the claims made in the introduction: that the probabilistic algorithm finds replicas quickly when they are nearby, that it fails quickly if they are not, and that this combination leads to a net performance improvement. We first justify our choice of update algorithm.

A. The Probabilistic Update Algorithm

Figure 8 shows the bandwidth used by each of the three update algorithms described in Section II-D. The bandwidth num-

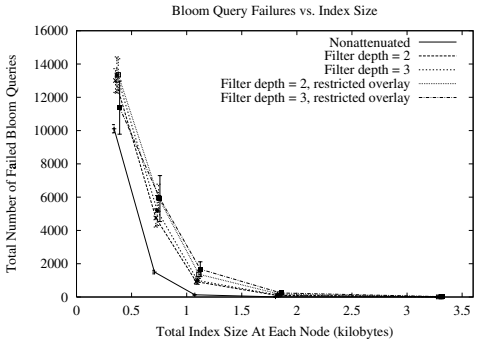


Fig. 9. *Bloom Query Failures vs. Index Size.* As the width of the bloom filters increases, the false positive rate drops quickly. Restricting the overlay network to minimize the number of physical interdomain edge crossings causes more false positives, but yields bandwidth advantages (see Figure 12). The restricted overlay has no effect on the false positives of nonattenuated filters.

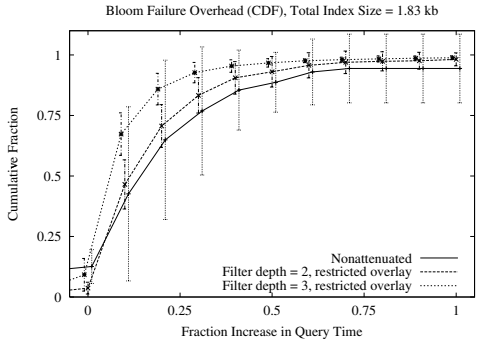
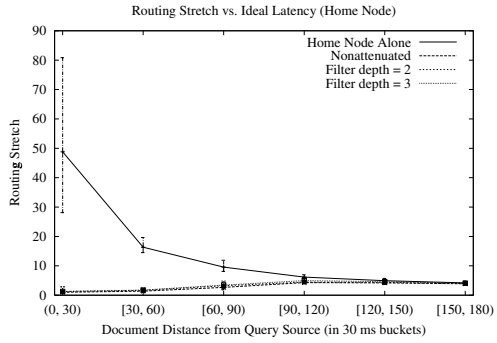
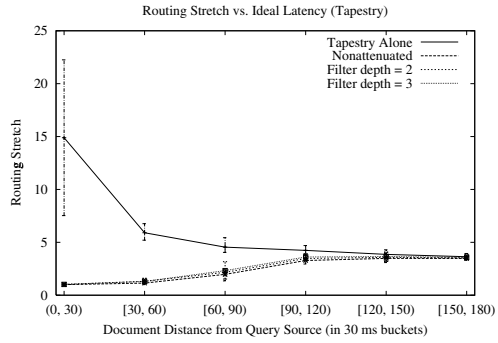


Fig. 10. *Additional Cost Due to Failed Bloom Queries.* Even when a probabilistic query fails, the total location time is not much more than if the query were handled by the deterministic algorithm alone. Here we see that for filters of depth 3, 86 percent of failed Bloom queries take only 20 percent longer than if they had been handled by the deterministic algorithm alone.



(a)



(b)

Fig. 11. *Routing Stretch vs. Ideal Latency.* Although there are few documents close to their query sources in the static experiments, the hybrid algorithm still manages to find them sufficiently quickly that it achieves a far lower routing stretch than the deterministic algorithms alone. The Home Node algorithm is shown in (a), and Tapestry is shown in (b). The error bars in this graph represent the 0th and 99th percentiles.

bers shown are for total number of bytes sent across any physical interdomain link in the system as the result of adding one document to a single server's cache.[4] As described in Section III, we measured these numbers at the end of each static test, dividing by the total number of documents added to produce an average cost per document. The graph clearly shows significant bandwidth reductions for the more advanced algorithms, so long as the attenuated Bloom filters being used are deep enough to take advantage of them. Destination filtering has no effect unless there are loops of length three or more in the update propagation graph, so no change is seen for depths one or two. Likewise, source filtering has no effect unless there are loops of length two or more, so no change is seen for depth one. The remainder of our experiments only use source filtering, since it either matches or outperforms the less advanced algorithms in every case.

B. Static Experiments

Our goal with the static experiments is to show first that the hybrid algorithm does not adversely affect the location of distant replicas, and second that it outperforms either deterministic algorithm alone in locating nearby ones.

[4]All error bars in our graphs represent the stability of the values shown with respect to changes in the underlying physical and overlay networks, and unless otherwise noted mark 95 percent confidence intervals.

To determine whether the hybrid algorithm would adversely affect queries for distant replicas, we graphed the number of hybrid queries which had to fall back on the deterministic algorithm. The result is shown in Figure 9. From this graph, we see that a total index size of around 1.83 kilobytes is sufficient to limit the number of such failing queries. Taking the number of documents per node times the average file size of the SURGE traces, we see that this index size is only 0.136 percent of the size of the data.

To further qualify the impact of failed Bloom queries, we show the cumulative distribution function of how much longer a failed Bloom query takes than one which had used Tapestry from the beginning in Figure 10. This graph shows that even when a probabilistic query fails, the total location time is not much more than if the query were handled by the deterministic algorithm alone. For example, with filters of depth 3, 86 percent of failed Bloom queries take only 20 percent longer than if they had been handled by the deterministic algorithm alone. Thus by using Bloom filters of a reasonable size, we incure only a limited number of failed queries, and the failures that do occur only minimally affect the total routing time.

Figure 11 shows the average routing stretch of the hybrid algorithm as a function of the query source's distance from the queried document. In Figure 11 (a), the deterministic algorithm used is home node routing, as described in Section III-A.1; in Figure 11 (b), Tapestry is used. In both cases, the total size of

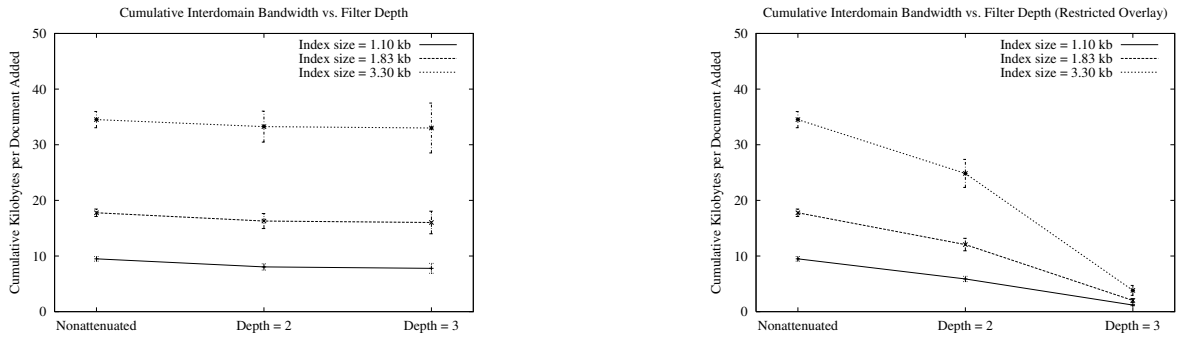


Fig. 12. *Interdomain Bandwidth vs. Filter Depth* These two graphs show the amount of interdomain bandwidth consumed by the update algorithm as a function of filter depth and overlay topology. In Figure (a), the overlay graph is constructed greedily, and all depths use roughly the same amount of bandwidth. In Figure (b), the number of overlay edges crossing a physical interdomain edge has been limited as much as possible while maintaining the average neighbor reachability per node. In this case, the topology discovery properties of the attenuated bloom filters greatly reduce update bandwidth.

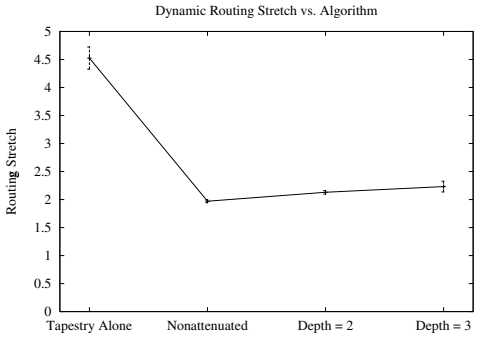


Fig. 13. *Dynamic Routing Stretch vs. Algorithm.* This graph shows the average routing stretch as a function of routing algorithm for the dynamic simulations. The hybrid algorithm far outperforms Tapestry alone for all filter depths. See the text for further discussion.

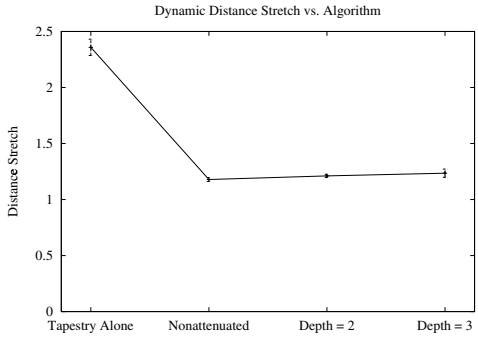


Fig. 14. *Dynamic Distance Stretch vs. Algorithm.* This graph shows the ratio of the distance between the query source and the replica which was actually located to the distance between the query source and closest replica available. The hybrid algorithm finds replicas which are closer to the query source on average than Tapestry alone.

the Bloom filter index at each node is fixed at 0.136 percent of the data size, as suggested by the previous results. One interesting aspect of these graphs is that the attenuated filters are providing comparable advantage to the nonattenuated ones using fewer immediate neighbors.

As described in the introduction, the closer the query source lies to the queried document, the less optimally the deterministic algorithms perform. The hybrid algorithm achieves a lower average stretch than either of the deterministic algorithms alone and reduces the variance of the stretch as well. Another interesting feature of Figure 11 is that Tapestry achieves a far lower routing stretch than home node location, especially for nearby replicas. This effect is produced by the locality inherent in the Tapestry routing mesh.

A final datum from the static experiments is shown in Figure 12, which graphs update bandwidth as a function of filter depth. From Figure 12 (a) we see that in a greedily-constructed overlay network, in which all nodes are connected to some number of their closest neighbors, attenuation does not provide any bandwidth advantages. However, Figure 12 (b) shows that if the number of overlay edges which traverse each physical interdomain edge is limited, the topology discovery features of source filtering greatly reduce the bandwidth consumed on those physical edges. Since these edges can easily become bottlenecks in real networks, we view this topology discovery property as a real benefit of the attenuated filters.

C. Dynamic Experiments

In contrast to the static experiments, the dynamic ones allow for the existence of multiple replicas of every document, subject to the constraints of the caching scheme described in Section III. Furthermore, the use of the SURGE traffic generator provides for substantial locality in the reference stream. As a result of these two factors, it is often the case in these experiments that several replicas exist near any given query source, so the hybrid algorithm has a far greater opportunity to improve performance than in the static experiments, where each document existed only on one node.

Figure 13 shows the average routing stretch in the dynamic experiments as a function of routing algorithm. In general, the hybrid algorithm far outperforms Tapestry alone, by as much as a factor of 2.1. Furthermore, Figure 14 shows the ratio of the distance between the query source and the replica which was actually located to the distance between the query source and closest replica available. Once again, the hybrid algorithm outperforms Tapestry alone, again by as much as a factor of 1.94. Not only does the hybrid algorithm find replicas in less time than Tapestry, it also finds closer replicas.

Our last graph, Figure 15, shows the total interdomain bandwidth consumed during the entire dynamic test case, measured as the total number of bytes that traverse all physical interdomain edges in the network. As mentioned above, the hybrid al-

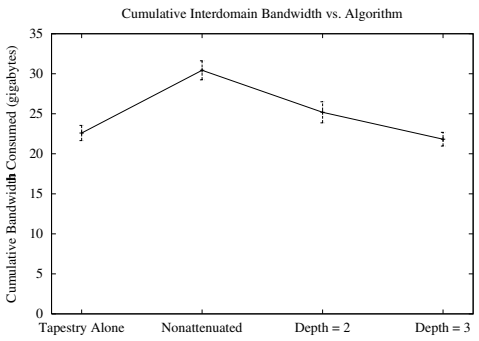


Fig. 15. *Dynamic Bandwidth Consumed vs. Algorithm.* The total interdomain bandwidth used by the hybrid algorithm is comparable to that used by Tapestry alone; the bandwidth reduction resulting from the closer replicas found by the hybrid algorithm offsets the increased bandwidth usage of the update algorithm. Furthermore, the bandwidth reduction gained by the topology discovery properties of the attenuated algorithm can be clearly seen.

gorithm finds replicas closer to the query source than Tapestry alone. As shown in Figure 15, the resulting reduction in bandwidth from this higher location quality is sufficient to mask the additional bandwidth being used by the hybrid algorithm during filter updates. Thus the improved performance of the hybrid algorithm does not imply a further cost in bandwidth.

V. Future Work

There are at least two ways in which this work could be improved. First, in our simulations we construct the Bloom filter overlay graphs using global knowledge. It seems reasonable to believe that simple overlay graphs could be constructed in a self-organizing manner; for instance, the Tapestry overlay is so constructed. However, as shown in Section IV-B, the bandwidth consumption of the attenuated Bloom filters can be dramatically reduced by placing restrictions on the structure of the overlay with respect to the underlying physical network. The design of algorithms to adhere to such restrictions while producing an overlay network in a self-organizing manner is thus an important component of our future work.

Second, since the caches in our system are managed in LRU order, every read causes at least one new data item to be published in the deterministic algorithm and propagated as a filter update in the probabilistic scheme. This caching policy obviously generates more update traffic than a more advanced one such as LRU-k [18] or n-chance forwarding [19] might. Since an update to a cache causes Tapestry to send only $O(\log N)$ messages, whereas the probabilistic algorithm must send some amount of information to every server in its filters' range, using these more advanced algorithms should only improve the bandwidth consumption of the probabilistic algorithm relative to Tapestry. Our current results are thus somewhat pessimistic with respect to the bandwidth usage of our algorithm.

VI. Related Work

Bloom filters [11] have long been used as a lossy summary technique. To our knowledge, however, we are the first to combine them into a compound, topology-aware data structure.

In [20], Bloom filters were used to improve the efficiency of distributed join operations by filtering elements without consuming network bandwidth. In [21], Aoki used Bloom filters to guide searches through generalized search trees.

Both the Summary Cache [12] and Cache Digests [22] use Bloom filters to to summarize the contents of a set of cooperating web caches. Both techniques are similar to our nonattenuated scheme, but use HTTP as their deterministic algorithm. In contrast to both schemes, we assume documents are highly mobile, requiring frequent update propagation; this frequency motivates our concern for update efficiency. In contrast to both Summary Cache and our work, the Cache Digest scheme polls for updates periodically rather than pushing them to neighbors as changes occur.

The Secure Discovery Service (SDS) [23] uses Bloom filters to route queries to appropriate *services*, such as printers or scanners; in that work, service attributes are arranged in a tree with the Bloom filters at each node summarizing the attributes of the node's children. Consequently, the accuracy of information decreases as a search climbs toward the root of the service tree, leading to wasted search traffic through the root node. In contrast, we use attenuated Bloom filters only for local-area routing, falling back on a bandwidth-efficient protocol in the wide area.

Our home node location protocol shares elements with existing directory services such as the Internet Domain Name Service (DNS) [13] and Globe [24]. Like our algorithms, DNS includes provisions for the caching of location information throughout the network, but does so using a weak consistency model that would not be desirable with objects moving at the frequently as we assume in this paper. The Globe system provides a hierarchical organization for replicas that might provide faster updates of location information than DNS. The three-hop location and routing protocol of the home node solution also resembles optimizations used in cache-coherent multiprocessors such as DASH [14].

The problem of constructing a practical location independent routing infrastructure has been tackled in several different projects. Although we chose Tapestry [10] in Section III-A, several competing architectures include CAN [7], Chord [8], Pastry [9]. All of these architectures provide guaranteed, deterministic routing from a client to a close replica. The exact details of the proposals are not particularly relevant to this paper, other than that they can serve as realistic fall-back algorithms for our probabilistic location techniques.

VII. Conclusion

In this paper we have presented a new, probabilistic routing algorithm designed to improve the location latency of existing deterministic approaches. The algorithm is based on a new data structure we call an attenuated Bloom filter. Our algorithm finds nearby replicas quickly, and if no such replicas exist, it fails quickly as well. Furthermore, we have shown that our algorithm may be combined with a deterministic algorithm to improve average routing stretch for nearby documents, where it matters the most. Finally, we have demonstrated that when replicas are allowed to move in response to a request stream modeled after real-world access patterns, this combination improved average performance by as much as a factor of 2.1. We are satisfied enough with our results that we are using this probabilistic algorithm as part of the routing subsystem of OceanStore.

```
PROCESSUPDATE (n, U, s)
1  F ← NEIGHBORFILTER (n)
2  foreach (r, c, v) ∈ U do F_rc ← v endfor
3  foreach n' ∈ N \ n do
4      L ← LASTUPDATEFILTER (n')
5      U' ← ∅
6      foreach (r, c, v) ∈ U do
7          r' ← r + 1
8          if v = 0 ∧ L_r'c = 1 then
9              γ ← true
10             foreach n'' ∈ N \ n' do
11                 F' ← NEIGHBORFILTER (n'')
12                 if F'_r'c = 1 then γ ← false endif
13             endfor
14             if γ then
15                 L_r'c ← 0
16                 U' ← U' ∪ {(r', c, 0)}
17             endif
18         elsif v = 1 ∧ L_r'c = 0 then
19             if ¬ IGNORINGSOURCE (n', s) then
20                 L_r'c ← 1
21                 U' ← U' ∪ {(r', c, 1)}
22             endif
23         endif
24     endfor
25     if U' ≠ ∅ then SENDUPDATE (n', U', s) endif
30 endfor
```

Fig. 16. *Pseudo-code for* PROCESSUPDATE. See text for description.

APPENDIX

This section presents the pseudo-code for the source filtering update algorithm. Its arguments are a neighbor number (n), an update (U), and the source of that update (s), where an update is a set of triples, (r, c, v), representing the intention to change the value in the neighbor's Bloom filter at row r and column c to value v. Lines 1–2 of the procedure look up the filter for the neighbor who sent the update (F) and apply the given changes. Then, for each other neighbor (n'), we look up our image of the last record they have of our documents (lines 3–4). This is an attenuated filter, L, identical in contents to the F they would look up for us if we sent them an update. For every cleared bit in the update, we check to see if this neighbor's filter holds a set bit. If so, we can only clear it if no other neighbor has the given bit set (lines 10–13). If we make a change, we include it in the outgoing update (lines 14–17). Next, if the incoming update has a bit set which is not set in the filter for n', and that neighbor is not ignoring this source, we change its filter and append the change to the outgoing update (lines 18–23). Finally, if we have changed anything for this neighbor, we send it the update we've constructed.

REFERENCES

[1] W. Bolosky, J. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs," in *Proc. of Sigmetrics*, June 2000.

[2] I. Clark, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000, pp. 311–320.

[3] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos, "Prototype implementation of archival intermemory," in *Proc. of IEEE ICDE*, Feb. 1996, pp. 485–495.

[4] J. Kubiatowicz et al., "Oceanstore: An architecture for global-scale persistent storage," in *Proc. of ASPLOS*. ACM, Nov. 2000.

[5] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, "Wide-area cooperative storage with CFS," in *Proc. of ACM SOSP*, October 2001.

[6] Peter Druschel and Antony Rowstron, "PAST: A large-scale, persistent peer-to-peer storage utility," in *Proc. of HOTOS Conf.*, 2001.

[7] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker, "A scalable content-addressable network," in *Proceedings of SIGCOMM*. ACM, August 2001.

[8] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of SIGCOMM*. ACM, August 2001.

[9] Peter Druschel and Antony Rowstron, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Proc. of ACM SOSP*, 2001.

[10] B. Zhao, A. Joseph, and J. Kubiatowicz, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB//CSD-01-1141, University of California, Berkeley Computer Science Division, April 2001.

[11] B. Bloom, "Space/time trade-offs in hash coding with allowable errors.," in *Communications of the ACM*, July 1970, vol. 13(7), pp. 422–426.

[12] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area Web cache sharing protocol," in *Proc. of ACM SIGCOMM Conf.*, Sept. 1998, pp. 254–265.

[13] P.V. Mockaptris and K. Dunlap, "Development of the domain name system," in *Proc. of ACM SIGCOMM Conf.*, August 1988.

[14] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy, "The dash prototype: Logic overhead and performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, pp. 41–61, January 1993.

[15] C. Plaxton, R. Rajaraman, and A. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *Proc. of ACM SPAA*, June 1997.

[16] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proc. of INFOCOMM*, 1996.

[17] Paul Barford and Mark Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proc. of Sigmetrics*, 1988.

[18] E. O'Neil, P. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," in *Proc. of ACM SIGMOD Conf.*, May 1993.

[19] M. Dahlin, T. Anderson, D. Patterson, and R. Wang, "Cooperative caching: Using remote client memory to improve file system performance.," in *Proc. of USENIX Symp. on OSDI*, Nov. 1994.

[20] Lothar F. Mackert and Guy M. Lohman, "R* optimizer validation and performance evaluation for distributed queries," in *Proc. of Intl. Conf. on VLDB*, August 1986.

[21] Paul M. Aoki, "Generalizing "search" in generalized search trees," in *Proc. 14th Int'l Conf. on Data Engineering*, February 1998.

[22] Alex Rousskov and Duane Wessels, "Cache digests," in *Proc of 3rd Int'l World Wide Web Caching Workshop*, 1998.

[23] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz, "An architecture for a secure service discovery service," in *Proc. of ACM/IEEE MobiCom Conf.*, 1999.

[24] M. van Steen, F.J. Hauck, G. Ballintijn, and A.S. Tanenbaum, "Algorithmic design of the globe wide-area location service.," *The Computer Journal*, vol. 41, no. 5, 1998.