# LOOKING UP DATA IN P2P SYSTEMS

Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica*
MIT Laboratory for Computer Science

## 1. Introduction

The recent success of some widely deployed peer-to-peer (P2P) file sharing applications has sparked new research in this area. We are interested in the P2P systems that have no centralized control or hierarchical organization, where the software running at each node is equivalent in functionality. Because these completely *decentralized* systems have the potential to significantly change the way large-scale distributed systems are built in the future, it seems timely to review some of this recent research.

The main challenge in P2P computing is to design and implement a robust distributed system composed of inexpensive computers in unrelated administrative domains. The participants in a typical P2P system might be home computers with cable modem or DSL links to the Internet, as well as computers in enterprises. Some current P2P systems have reported tens of thousands of simultaneously active participants, with half a million participating machines over a week-long period.

P2P systems are popular and interesting for a variety of reasons:

1. The barriers to starting and growing such systems are low, since they usually don't require any special administrative or financial arrangements, unlike with centralized facilities.

2. P2P systems suggest a way to aggregate and make use of the tremendous computation and storage resources that otherwise just sit idle on computers across the Internet when they are not in use.

3. The decentralized and distributed nature of P2P systems gives them the potential to be robust to faults or intentional attacks, making them ideal for long-term storage as well as for lengthy computations.

P2P computing raises many interesting research problems in distributed systems. In this short paper we will look at one of them, *the lookup problem*: How do you find any given data item in a large P2P system in a scalable manner, without any centralized servers or hierarchy? This problem is at the heart of any P2P system. It is not addressed well by most systems in popular use, and it provides a good example of how the challenges of designing P2P systems can be addressed.

The recent algorithms developed by several research groups for the lookup problem present a simple and general interface, a *distributed hash table* (DHT). Data items are inserted in a DHT and found by specifying a unique *key* for that data. To implement a DHT, the underlying algorithm must be able to determine which node is responsible for storing the data associated with any given key. To solve this problem, each node maintains information (e.g., the IP address) of a small number of other nodes ("neighbors") in the system, forming an *overlay network* and routing messages in the overlay to store and retrieve keys.

One might believe from recent news items that P2P systems are good for illegal music swapping and little else, but this would be a rather hasty conclusion. The distributed hash table, for example, is increasingly finding uses in the design of robust, large-scale distributed applications. It appears to provide a general-purpose interface for location-independent naming upon which a variety of applications can be built. Furthermore, distributed applications that make use of such an infrastructure inherit robustness, ease of operation, and scaling properties. A significant amount of research effort is now being devoted to investigating these ideas.

## 2. The lookup problem

The lookup problem is simple to state. Some publisher inserts an item $X$, say a file, in the system. At a later point in time, some consumer wants to retrieve $X$, in general when the publisher isn't online and connected to the system. How does the consumer find the location of a server that has a replica of $X$?

One approach is to maintain a central database that maps a file name to the locations of servers that store the song. Napster (`http://www.napster.com/`) adopts this approach for song titles, but it has inherent reliability and scalability problems that make it vulnerable to attacks on the database.

Another approach, at the other end of the spectrum, is for the consumer to broadcast a message to all its neighbors with a request for $X$. When a node receives such a request, it checks its local database. If it has $X$, it responds with the item. Otherwise, it forwards the request to its neighbors, which execute the same protocol. Gnutella (`http://gnutella.wego.com/`) has a protocol in this style with some mechanisms to avoid request loops. However, this "broadcast" approach doesn't scale either [7], because of the bandwidth consumed by broadcast messages and the compute cycles consumed by the many nodes that must handle these messages. In fact, the day after Napster was shut down, reports indicate that the Gnutella network collapsed under its own load, created when a large number of users migrated to using it for sharing music.

To reduce the cost of broadcast messages, one can organize the

* University of California, Berkeley.

nodes in the network into a hierarchy, like the Internet's Domain Name System (DNS) does. Searches start at the top of the hierarchy and, by following forwarding references from node to node, traverse a single path down to the node that contains the desired data. Directed traversal of a single path consumes fewer resources than a broadcast. Many of the current popular systems, such KaZaA, Grokster, and MusicCity Morpheus, which are all based on FastTrack's P2P platform (`http://www.fasttrack.nu/`), adopt this approach. The disadvantage of the hierarchical approach is that the nodes higher in the tree take a larger fraction of the load than the leaf nodes, and therefore require more expensive hardware and more careful management. The failure or removal of the tree root or a node sufficiently high in the hierarchy can be catastrophic.

*Symmetric* distributed lookup algorithms avoid the drawbacks of the previous approaches. Searches are carried out by following references from node to node until the appropriate node containing the data is found. Unlike the hierarchy, no node plays a special role—a search can start at any node, and each node is involved in only a small fraction of the search paths in the system. As a result, no node consumes an excessive amount of resources while supporting searches. These new algorithms are designed to scale well—they require each node to only maintain information about a small number of other nodes in the system, and they allow the nodes to self-organize into an efficient overlay structure with little effort.

Freenet [2] was one of the first algorithms in this class. Queries are forwarded from node to node until the desired object is found. But a key Freenet objective, anonymity, creates some challenges for the system. To provide anonymity, Freenet avoids associating a document with any predictable server, or forming a predictable topology among servers. As a result, unpopular documents may simply disappear from the system, since no server has the responsibility for maintaining replicas. A search may often need to visit a large fraction of the Freenet network [7].

The recent crop of P2P lookup algorithms does not pose anonymity as a primary objective (though it remains an interesting goal, and we believe that these new algorithms provide useful infrastructure to support it). This simplification lets them offer *guarantees* that that data can be reliably found in the system.

The rest of this article discusses four recent P2P lookup algorithms that have provable guarantees: CAN, Chord, Pastry, and Tapestry. These algorithms stress the ability to scale well to large numbers of nodes, to locate keys with low latency, to handle node arrivals and departures scalably, to ease the maintenance of per-node routing tables, and to balance the distribution of keys evenly amongst the participating nodes.

## 3. Distributed hash table (DHT)

A hash-table interface is an attractive foundation for a distributed lookup algorithm because it places few constraints on the structure of keys or the data they name. The main requirements are that data be identified using unique numeric keys, and that nodes be willing to store keys for each other. This organization is in contrast to Napster and Gnutella, which search for keywords, and assume that data is primarily stored on the publisher's node. However, such systems could still benefit from a distributed hash table—for example, Napster's centralized database recording the mapping between nodes and songs could be replaced by a distributed hash table.

A DHT implements just one operation: `lookup(key)` yields the identity (e.g., IP address) of the node currently responsible for the given key. A simple distributed storage application might use this interface as follows. Someone who wants to publish a file under a particular unique name would convert the name to a numeric key using an ordinary hash function such as SHA-1, then call `lookup(key)`. The publisher would send the file to be stored at the resulting node. Someone wishing to read that file would obtain its name, convert it to a key, call `lookup(key)`, and ask the resulting node for a copy of the file. A complete storage system would have to take care of replication, caching, authentication, and other issues; these are outside the immediate scope of the lookup problem.

To implement DHT's, lookup algorithms have to address the following issues:

**Mapping keys to nodes in a load-balanced way.** All algorithms do this in essentially the same way. Both nodes and keys are mapped using a standard hash function into a string of digits. (In Chord, the digits are binary. In CAN, Pastry, and Tapestry, the digits are in a higher order base.) A key hashing to a given digit string $s$ is then assigned to the node with the "closest" digit string to $s$ (e.g., the one that is the closest numeric successor to $s$, or the one with the longest matching prefix).

**Forwarding a lookup for a key to an appropriate node.** Any node that receives a query for a given key identifier $s$ must be able to forward it to a node whose ID is "closer" to $s$. This will guarantee that the query eventually arrives at the closest node. To achieve this, each node maintains a routing table with entries for a small number of carefully chosen other nodes. If the key ID is larger than the current node, we can forward to a node that is larger than the current node but still smaller than the key, thus getting numerically closer (a converse rule holds if the key ID is smaller than the current node ID). Alternatively, we can forward to a node whose ID has more digits "correct" (that is, in common with the key ID). For example, if a node has ID $7634$ and a key has ID $7892$, then forwarding to node $7845$ brings an additional digit into agreement with the key ID.

**Building routing tables.** To forward lookup messages, each node needs to know about some other nodes. To support the first forwarding rule of getting numerically closer to the key ID, each node should know of its *successor*—the node with the closest succeeding ID. This successor is a valid forwarding node for any key ID that is larger than the current node. To support the second forwarding rule of correcting digits left to right, each node should be aware of other nodes whose identifers match in some prefix.

There are substantial differences between how the algorithms build and maintaining their routing tables as node join and leave. The discussion of each algorithm is organized around these issues.

## 4. CAN

CAN [9] uses a $d$-dimensional Cartesian coordinate space (for some fixed $d$) to implement the DHT abstraction. The coordinate space is partitioned into hyper-rectangles, called zones. Each node in the system is responsible for a zone, and a node is identified by the boundaries of its zone. A key is mapped onto a point in the coordinate space, and it is stored at the node whose zone contains the point's coordinates. Figure 1(a) shows a 2-dimensional $[0, 1] \times [0, 1]$ CAN with six nodes.

Each node maintains a routing table of all its neighbors in coordinate space. Two nodes are neighbors if their zones share a $d-1$ dimensional hyperplane.

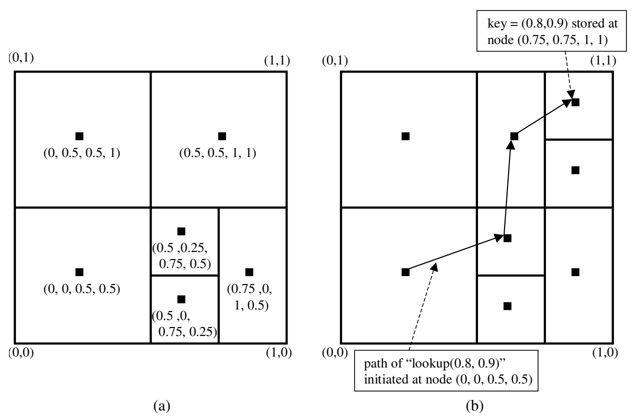


**Figure 1: (a) A 2-dimensional CAN with 6 nodes. The coordinate space wraps (not illustrated here). Each node is assigned a zone, and a node is identified by the boundaries of its zone. (b) The path followed by the lookup for key $(0.8, 0.9)$. The lookup is initiated at node $(0, 0, 0.5, 0.5)$.**

The lookup operation is implemented by forwarding the query message along a path that approximates the straight line in the coordinate space from the querier to the node storing the key. Upon receiving a query, a node forwards it to the neighbor closest in the coordinate space to the node storing the key, breaking ties arbitrarily. Figure 1(b) shows the path followed by the lookup for key $(0.8, 0.9)$. Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$. If $d = O(\log N)$, CAN lookup times and storage needs match the other protocols surveyed in this paper.

To join the network, a new node first chooses a random point $P$ in the coordinate space, and asks a node already in the network to find the node $n$ whose zone contains $P$. Node $n$ splits its zone in half and assigns one of the halves to the new node. The new node can easily initialize its routing table, since all its neighbors, excepting $n$ itself, are amongst $n$'s neighbors. Once it has joined, the new node announces itself to its neighbors. This allows the neighbors to update their routing tables with the new node.

When a node departs, it hands its zone to one of its neighbors. If merging the two zones creates a new valid zone, the two zones are combined into a larger zone. If not, the neighbor node will temporarily handle both zones. If a node fails, CAN implements a protocol that allows the neighbor with the smallest zone to take over. One potential problem is that multiple failures will result in the fragmentation of the coordinate space, with some nodes handling a large number of zones. To address this problem, CAN runs a special node-reassignment algorithm in background. This algorithm tries to assign zones that can be merged into a valid zone to the same node, and then combine them.

CAN proposes a variety of techniques that reduce the lookup latency. In one such technique, each node measures the network round-trip time (RTT) to each of its neighbors. Upon receiving a lookup, a node forwards the message to the neighbor with the maximum ratio of the progress towards the destination node in the coordinate space to the RTT.

Another technique, called multiple realities, uses multiple coordinate spaces to simultaneously improve both lookup latency and robustness of CAN. Here, each node is assigned a different zone in each coordinate space. The keys are replicated in each space, improving robustness to node failure. To forward a query message, a node checks its neighbors in each reality and forwards the message to the closest one.

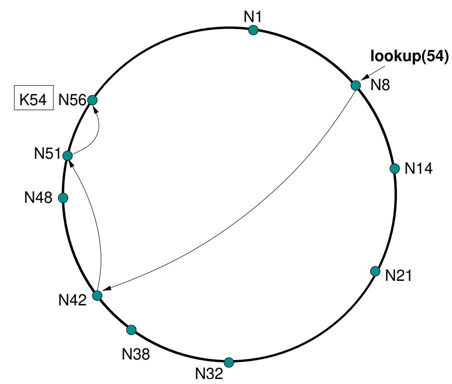


**Figure 2: The path of a Chord lookup for key 54, initiated by node 8, in an ID space of size $2^6$. Each arrow represents a message forwarding a lookup through one of a node's finger table entries. For example, node 8 forwards to node 42 because node 42 is the first node halfway around the ID space from 8.**

To improve load balance, CAN uses a simple technique that achieves a more uniform space partitioning. During the join operation, a node checks its neighboring zones before splitting its zone. If some neighbors have larger zones, the joining node asks the neighbor with the largest zone to split instead.

CAN is being used to develop a scalable application-level multicast protocol [10], and a chat application on top of Sun's JXTA technology (see `http://jxme.jxta.org/Demo.html`).

## 5. Chord

Chord [14] assigns ID's to both keys and nodes from the same one-dimensional ID space. The node responsible for key $k$ is called its *successor*, defined as the node whose ID most closely follows $k$. The ID space wraps around to form a circle, so ID 0 follows the highest ID.

Chord performs lookups in $O(\log N)$ time, where $N$ is the number of nodes, using a per-node *finger table* of $\log N$ entries. A node's finger table contains the IP address of a node halfway around the ID space from it, a quarter-of-the-way, and so forth in powers of two. A node forwards a query for key $k$ to the node in its finger table with the highest ID less than $k$. The power-of-two structure of the finger table ensures that the node can always forward the query at least half of the remaining ID-space distance to $k$. As a result Chord lookups use $O(\log N)$ messages.

Chord ensures correct lookups despite node failures using a *successor list*: each node keeps track of the IP addresses of the next $r$ nodes immediately after it in ID space. This allows a query to make incremental progress in ID space even if many finger table entries turn out to point to crashed nodes. The only situation in which Chord cannot guarantee to find the current live successor to a key is if *all* $r$ of a node's immediate successors fail simultaneously, before the node has a chance to correct its successor list. Since node ID's are assigned randomly, the nodes in a successor list are likely to be unrelated, and thus suffer independent failures. In such a case, relatively small values of $r$ (such as $\log N$) make the probability of simultaneous failure vanishingly small.

A new node $n$ finds its place in the Chord ring by asking any existing node to look-up $n$'s ID. All that is required for the new node to participate correctly in lookups is for it and its predecessor to update their successor lists. Chord does this in a way that ensures correctness even if nodes with similar ID's join concurrently. The new node, and existing nodes, will have to update their finger tables; this happens in the background because it is only required for performance, not correctness. The new node must also acquire whatever data is associated with the keys it is responsible for; the successor relationship ensures that all of these keys may be fetched from the new node's successor.

Randomness ensures that Chord key and node ID's are spread roughly uniformly in ID space, ensuring approximately balanced load among nodes. Chord allows control over the fraction of the key space stored on a node by means of *virtual nodes*. Each physical node can participate in the Chord system with multiple virtual node ID's; increasing the number of such ID's will increase the number of keys the node must store.

Chord can route queries over low-latency paths by taking advantage of the fact that multiple finger table entries are typically available to take a query closer to its destination. Each such entry has a known cost (latency in the underlying net) and benefit (progress in ID space); Chord incorporates heuristics to trade-off between these.

The main emphasis in Chord's design is robustness and correctness, achieved by using simple algorithms with provable properties even under concurrent joins and failures. Chord is in use as a part of the experimental CFS [3] wide-area file store, and as part of the Twine [1] resource discovery system. A public distribution of Chord can be found at `http://www.pdos.lcs.mit.edu/chord/`.

## 6. Pastry

Pastry [11] gives each node has a randomly chosen ID, which conceptually indicates its position on an identifier circle. Pastry routes messages with a key to the live node with node ID numerically closest to the key. The digits in the ID space are in base $2^b$, where $b$ is an algorithm parameter typically set to 4 with 128-bit ID's.

Pastry uses a prefix-based forwarding scheme. Each node $n$ maintains a *leaf set* $L$, which is the set of $|L|/2$ nodes closest to $n$ and larger than $n$, and the set of $|L|/2$ nodes closest to $n$ and smaller than $n$. The correctness of this leaf set is the only thing required for correct forwarding; correct forwarding occurs unless $|L|/2$ nodes with adjacent IDs fail simultaneously. The leaf set is conceptually similar to Chord's successor list.

To optimize forwarding performance, Pastry maintains a *routing table* of pointers to other nodes spread in the ID space. A convenient way to view this information is as $\lceil \log_{2^b} N \rceil$ rows, each with $2^b - 1$ entries each. Each entry in row $i$ of the table at node $n$ points to a node whose ID shares the first $i$ digits with node $n$, *and* whose $i + 1$st digit is different (there are at most $2^b - 1$ such possibilities).

Given the leaf set and the routing table, each node $n$ forwards queries as follows. If the sought key is covered by $n$'s leaf set, then the query is forwarded to that node. In general, of course, this will not occur until the query reaches a point close to the key's ID. In this case, the request is forwarded to a node from the routing table that has a longer shared prefix (than $n$) with the sought key.

Sometimes, the entry for such a node may be missing from the routing table or that node may be unreachable from $n$. In this case, $n$ forwards the query to a node whose shared prefix is at least as long as $n$'s, *and* whose ID is numerically closer to the key. Such a node must clearly be in $n$'s leaf set unless the query has already arrived at the node with numerically closest ID to the key, or at its immediate neighbor.

If the routing tables and leaf sets are correct, the expected number of hops taken by Pastry to route a key to the correct node is at most $\lceil \log_{2^b} N \rceil$. This is because each step via the routing table increases the number of common digits between the node and the key by 1, and reduces the set of nodes with whose IDs have a longer prefix match by $2^b$. Finally, once the query hits in the leaf set, only one more hop is needed.

Pastry has a join protocol that attempts to build the routing tables and leaf sets by obtaining information from nodes along the path from the bootstrapping node and the node closest in ID space to the new node. It may be simplified by maintaining the correctness of the leaf set for the new node, and building the routing tables in the background, as in Chord. This approach is used in Pastry when a node leaves; only the leaf sets of nodes are immediately updated, and routing table information is corrected only on demand when a node tries to reach a non-existent one and detects that it is unavailable.

Finally, Pastry implements heuristics to route queries according to a proximity metric. It is likely to forward a query to the nearest (in the network sense) one among $k$ possible nodes. To do this, each node maintains a *neighborhood set* of other nodes nearest to it accoding to a network-proximity metric (e.g., number of IP hops). The routing table is initialized such that each entry refers to a "nearby" node with the appropriate prefix, among all such live nodes.

Pastry is being used to develop two P2P services: PAST, a storage facility [12], and SCRIBE, an event notification facility. More information is available at `http://www.cs.rice.edu/CS/Systems/Pastry/`.

## 7. Tapestry

Tapestry [4] is based on a lookup scheme developed by Plaxton et al. [8]. Like all the other schemes, Tapestry maps node and key identifiers into strings of numbers, and forwards lookups to the correct node by "correcting" a single digit at a time. For example, if node number 723948 receives a query for item number 723516, whose first 3 digits match the node's identifier, it can forward that query to a "closer" node such as 723584, which has 4 initial digits matching the query. To support this forwarding, a node needs to keep track, for each prefix of its own identifier, of nodes that match that prefix but differ in the next digit. For example, node 723948 would keep track of nodes with prefixes 71, 73, 74, and so on to allow it to correct the second digit of a query, of nodes with prefixes 721, 722, 724, 725, and so on to let it correct the third digit of a query, and so on.

What perhaps most distinguishes Tapestry from the other approaches we discuss is its focus on *proximity*. As a query hops around the network, it is preferable for that query to visit nodes close (in the underlying network) to the node doing the query, as this will reduce the latency involved in performing the query. If there are multiple copies of a desired item, it would be desirable for the closest copy to be found. Under certain models of the topology of the network, it can be proven that the Tapestry scheme will indeed find an approximately nearest copy of any sought key.

The gain in proximity comes at the cost of increased complexity. Because of the sensitivity of its proximity data structures, the original Tapestry data structure, which worked very well in a static environment, was unable to support dynamic joins and departures of nodes. Later versions added support for such dynamic operations, but the emphasis on proximity makes them complex. In contrast, other systems such as Chord and Pastry focus initially on ensuring that data items are found, ignoring cost. This allows for relatively simple lookup, join, and leave operations. Heuristics can then be placed on top of the simple operations in an attempt to support proximity search. The question of how central a role proximity should play in a P2P system remains open.

## 8. Summary and open questions

| | CAN | Chord | Pastry | Tapestry |
|---|---|---|---|---|
| Node state | $d$ | $\log N$ | $\log N$ | $\log N$ |
| Lookup | $dN^{1/d}$ | $\log N$ | $\log N$ | $\log N$ |
| Join | $dN^{1/d} + d\log(N)$ | $\log^2 N$ | $\log^2 N$ | $\log^2 N$ |

**Figure 3: Summary of the costs of the algorithms. The Per-node State line indicates how many other nodes each node knows about. The Lookup and Join lines indicate how many messages (i.e. Internet packets) are required for each operation. $N$ is the total number of nodes in the system. $d$ is CAN's number-of-dimensions parameter. All figures indicate asymptotic performance.**

The lookup algorithms described here are all still in development. Their current strengths and weaknesses reflect the designers' initial decisions about the relative priorities of different issues, and to some extent decisions about what to stress when publishing algorithm descriptions. Some of these issues are summarized below to help contrast the algorithms and highlight areas for future work.

**Operation costs.** Figure 3 summarizes the costs of fundamental operations. Chord, Pastry, and Tapestry are identical to within a constant factor. On the other hand, CAN's routing tables have constant size regardless of the system size, though the lookup cost grows faster than in the other systems. A key area for future work and evaluation is the effect of relatively frequent node joins and departures in large systems; even relatively modest costs for these operations could end up dominating overall performance.

**Fault tolerance and concurrent changes.** Most of the algorithms assume single events when considering the handling of nodes joining or failing out of the system. Chord and recent work on Tapestry also guarantee correctness for the difficult case of concurrent joins by nodes with similar IDs, as well as for simultaneous failures. Some recent research focuses on algorithms that improve efficiency under failures by avoiding timeouts to detect failed nodes [5, 6, 13].

**Proximity routing.** Pastry, CAN, and Tapestry have heuristics to choose routing table entries that refer to nodes that are nearby in the underlying network; this decreases the latency of lookups. Chord chooses routing table entries obliviously, so it has limited choice when trying to choose low-delay paths. Since a lookup in a large system could involve tens of messages, at dozens of milliseconds per message, reducing latency may be important. More work will likely be required to find latency reduction heuristics that are effective on the real Internet topology.

**Malicious nodes.** Pastry uses certificates secured in smart cards to prove node identity, which allows strong defenses against malicious participants. The cost, however, is trust in a central certificate authority. All of the algorithms described can potentially perform cross-checks to detect incorrect routing due to malice or errors, since it is possible to verify whether progress in the ID space is being made. Authenticity of data can be ensured cryptographically, so the worst a malicious node can do is convincingly deny that data exists. The tension between the desire to avoid restricting who can participate in a P2P system and the desire to hold participants responsible for their behavior appears to be an important practical consideration.

**Indexing and keyword search.** The distributed hash table algorithms outlined here retrieve data based on a unique identifier. In contrast, the widely deployed P2P file sharing services, such as Napster and Gnutella, are based on keyword search. While it is expected that distributed indexing and keyword lookup can be layered on top of the distributed hash model, it is an open question if indexing can be done efficiently.

In summary, these P2P lookup systems have many aspects in common, but comparing them also reveals a number of issues that will need further investigation or experimentation to resolve. They all share the DHT abstraction in common, and this has been shown to be beneficial in a range of distributed P2P applications. With more work, DHT's might well prove to be a valuable building block for robust, large-scale distributed applications on the Internet.

## References


[1] Balazinska, M., Balakrishnan, H., and Karger, D. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proc. Intl. Conf. on Pervasive Computing* (Zurich, Switzerland, Aug. 2002).

[2] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. ICSI Workshop on Design Issues in Anonymity and Unobservability* (Berkeley, California, June 2000). `http://freenet.sourceforge.net`.

[3] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001).

[4] Hildrum, K., Kubatowicz, J. D., Rao, S., and Zhao, B. Y. Distributed Object Location in a Dynamic Network. In *Proc. 14th ACM Symp. on Parallel Algorithms and Architectures* (Aug. 2002).

[5] Malkhi, D., Naor, M., and Ratajczak, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of Principles of Distributed Computing (PODC 2002)* (July 2002).

[6] Maymounkov, P., and Mazieres, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002).

[7] Oram, A., Ed. *Peer-to-Peer: Harnessing the Power of Disruptive Computation*. O'Reilly & Associates, 2001.

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

[9] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *Proc. ACM SIGCOMM* (San Diego, CA, August 2001), pp. 161–172.

[10] Ratnasamy, S., Handley, M., Karp, R., and Shenker, S. Application-level Multicast using Content-Addressable Networks. In *Proceedings of NGC 2001* (Nov. 2001).

[11] Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-s cale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)* (Nov. 2001).

[12] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Oct. 2001).

[13] Saia, J., Fiat, A., Gribble, S., Karlin, A., and Saroiu, S. Dynamically fault-tolerant content addressable networks. In *Proc. 1st International Workshop on Peer-to-Peer systems* (Mar. 2002).

[14] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM* (San Diego, Aug. 2001).