LOOKING UP DATA *in P2P Systems*

BY HARI BALAKRISHNAN, M. FRANS KAASHOEK, DAVID KARGER, ROBERT MORRIS, AND ION STOICA

The main challenge in P2P computing is to design and implement a robust and scalable distributed system composed of inexpensive, individually unreliable computers in unrelated administrative domains. The participants in a typical P2P system might include computers at homes, schools, and businesses, and can grow to several million concurrent participants.

P2P systems are attractive for several reasons:

- The barriers to starting and growing such systems are low, since they usually don't require any special administrative or financial arrangements, unlike centralized facilities;
- P2P systems offer a way to aggregate and make use of the tremendous computation and storage resources on computers across the Internet; and
- 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 article 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 popular systems currently in 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

Designing and implementing a robust distribution system composed of inexpensive computers in unrelated administrative domains.

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 (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 mainly used for illegal music-swapping and little else, but this would be a rather hasty conclusion. The DHT abstraction 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 (Project IRIS, a multi-institution, large-scale effort, is one example; see www.project-iris.net).

One might believe P2P systems are mainly used for illegal music-swapping and little else, but this would be a rather hasty conclusion.

The Lookup Problem

The lookup problem is simple to state: Given a data item *X* stored at some dynamic set of nodes in the system, find it. This problem is an important one in many distributed systems, and is the critical common problem in P2P systems.

One approach is to maintain a central database that maps a file name to the locations of servers that store the file. Napster (www.napster.com) adopted this approach for song titles, but this approach has inherent scalability and resilience problems: the database is a central point of failure.

The traditional approach to achieving scalability is to use hierarchy. The Internet's Domain Name System (DNS) does this for name lookups. 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 containing the desired data. The disadvantage of this approach is that failure or removal of the root or a node sufficiently high in the hierarchy can be catastrophic, and the nodes higher in the tree take a larger fraction of the load than the leaf nodes.

These approaches are all examples of structured lookups, where each node has a well-defined set of information about other nodes in the system. The advantage of structured lookup methods is that one can usually make guarantees that data can be reliably found in the system once it is stored.

To overcome the resilience problems of these schemes, some P2P systems developed the notion of symmetric lookup algorithms. Unlike the hierarchy, no node is more important than any other node as far as the lookup process is concerned, and each node is typically involved in only a small fraction of the search paths in the system. These schemes allow the nodes to self-organize into an efficient overlay structure.

At one end of the symmetric lookup spectrum, the consumer broadcasts 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 contains *X*, it responds with the item. Otherwise, it forwards the request to its neighbors, which execute the same protocol. Gnutella (gnutella.wego.com) has a protocol in this style with some mechanisms to avoid request loops. However, this "broadcast" approach doesn't scale well 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 the Gnutella network collapsed under the load created by a large number of users who migrated to it for sharing music.

One approach to handling such scaling problems is to add "superpeers" in a hierarchical structure, as is done in FastTrack's P2P platform (www.fasttrack.nu), and has been popularized by applications like KaZaA (www.kazaa.com). However, this comes at the expense of resilience to failures of superpeers near the top of the hierarchy. Furthermore, this approach does not provide guarantees on object retrieval.

Freenet [1] uses an innovative symmetric lookup strategy. Here, queries are forwarded from node to node until the desired object is found based on unstructured routing tables dynamically built up using caching. 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. Furthermore, a search may often need to visit a large fraction of nodes in the system, and no guarantees are possible.

The recent crop of P2P algorithms, including

CAN [8], Chord [11], Kademlia [6], Pastry [9], Tapestry [2], and Viceroy [5] are both structured and symmetric, unlike all the other systems mentioned here. This allows them to offer guarantees while simultaneously not being vulnerable to individual node failures. They all implement the DHT abstraction.

The rest of this article discusses these recent algorithms, highlighting design points and trade-offs. These algorithms incorporate techniques that scale well to large numbers of nodes, to locate keys with low latency, to handle dynamic node arrivals and departures, to ease the maintenance of per-node routing tables, and to balance the distribution of keys evenly among the participating nodes.

A Distributed Hash Table

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 values 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. The values could be actual data items (file blocks), or could be pointers to where the data items are currently stored.

A DHT implements just one operation: `lookup(key)` yields the network location of the node currently responsible for the given key. A simple distributed storage application might use this interface as follows. To publish a file under a particular unique name, the publisher would convert the name to a numeric key using an ordinary hash function such as SHA-1, then call `lookup(key)`. The publisher would then send the file to be stored at the node(s) responsible for the key. A consumer wishing to read that file would later obtain its name, convert it to a key, call `lookup(key)`, and ask the resulting node for a copy of the file.

To implement DHTs, lookup algorithms have to address the following issues:

Mapping keys to nodes in a load-balanced way. In general, all keys and nodes are identified using an m-bit number or identifier (ID). Each key is stored at one or more nodes whose IDs are "close" to the key in the ID space.

Forwarding a lookup for a key to an appropriate node. Any node that receives a query for a key identifier s must be able to forward it to a node whose ID is "closer" to s. This rule will guarantee that the query eventually arrives at the closest node.

Distance function. The two previous issues allude to the "closeness" of keys to nodes and nodes to each other; this is a common notion whose definition depends on the scheme. In Chord, the closeness is the numeric difference between two IDs; in Pastry and Tapestry, it is the number of common prefix bits; in Kademlia, it is the bit-wise exclusive-or (XOR) of the two IDs. In all the schemes, each forwarding step reduces the closeness between the current node handling the query and the sought key.

Building routing tables adaptively. To forward lookup messages, each node must know about some other nodes. This information is maintained in routing tables, which must adapt correctly to asynchronous and concurrent node joins and failures.

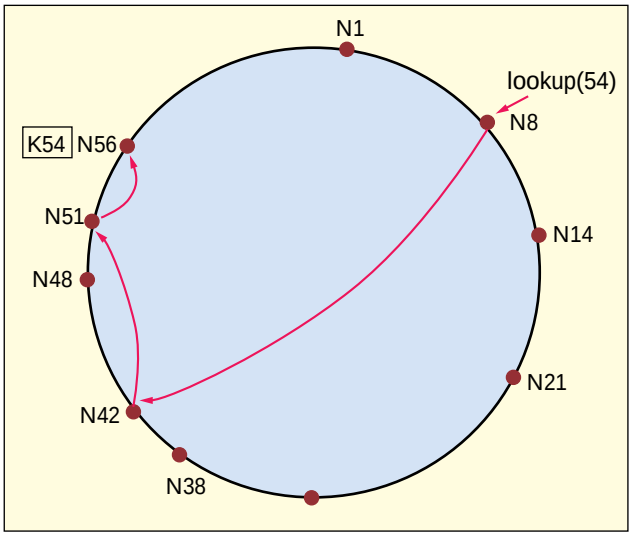

Figure 1. A structure resembling a skiplist data structure.

Routing in One Dimension

A key difference in the algorithms is the data structure that they use as a routing table to provide O(log N) lookups. Chord maintains a data structure that resembles a skiplist. Each node in Kademlia, Pastry, and Tapestry maintains a tree-like data structure. Viceroy maintains a butterfly data structure, which requires information about only constant other number nodes, while still providing O(log N) lookup. A recent variant of Chord uses de Bruijn graphs, which requires each node to know only about two other nodes, while also providing O(log N) lookup. We illustrate the issues in routing using Chord and Pastry's data structure.

Chord: Skiplist-like routing

Each node in Chord [11] has a finger table containing the IP address of a node halfway around the ID space from it, a quarter-of-the-way, an eighth-of-the-way, and so forth, in powers of two, in a structure that resembles a skiplist data structure (see Figure 1). A node forwards a query for key k to the node in its

finger table with the highest ID not exceeding k; the ID of this node is called the successor of 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, leading to O(log N) messages to resolve a query.

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 ensures correct lookups in the face of node failures and arrivals using a successor list: each node keeps track of the IP addresses of the next r nodes immediately after it in ID space. This solution allows a query to make incremental progress in ID space even if many finger-table entries turn out to point to failed or nonexistent 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 IDs are assigned randomly, the nodes in a successor list are likely to be unrelated, and thus suffer independent failures. Hence, for relatively small values of r (such as log N) the probability of simultaneous failure goes down to $1/N$.

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 IDs 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 all these keys may be fetched from the new node's successor.

Chord repairs its successor list and finger tables continuously using simple stabilization protocols. For instance, each node n periodically contacts its successor $s(n)$ and asks $s(n)$ for its predecessor. If the returned predecessor is not n, then the appropriate local corrections can be done.

Tree-like routing

Each node in the tree-based algorithms records, for each prefix, the location of some node with that prefix. Thus, each node knows a node with a prefix 0, 1, 00, 01, 10, 11, 000, and so forth. Pastry [9], Tapestry [2], and Kademlia [6] are examples of algorithms that use a tree-like data structure.

Pastry gives each node a randomly chosen ID, indicating its position on an identifier circle. It routes messages with a key to the live node with a node ID numerically closest to the key, using 128-bit IDs in base 2^b, where b is an algorithm parameter typically set to 4.

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 requirement for correctness; forwarding is always correct, unless $|L|/2$ nodes with adjacent IDs fail simultaneously.

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 $[\log_{2^b} N]$ 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 implements the forwarding step 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, it will not be, 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 because the node doesn't exist, or that node may be unreachable from n. In this case, n forwards the query to a node whose shared prefix with the key is at least as long as n's shared prefix with the key, 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 $[\log_{2^b} N]$.

Pastry has a join protocol that builds 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. 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 nonexistent one and detects that it is unavailable.

Pastry implements heuristics to route queries

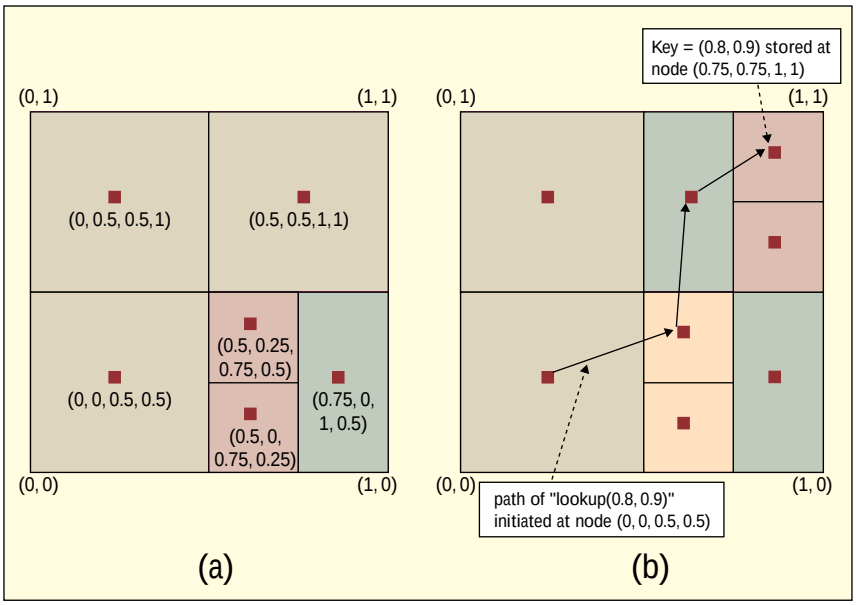

Figure 2a. A 2-dimensional [0,1] x [0,1] CAN with suy nodes.
Figure 2b. Breaking ties arbitrarily.

according to a network-proximity metric. Each node is likely to forward a query to the nearest one of k possible nodes, using a neighborhood set of other nearby nodes.

Routing in Multiple Dimensions

CAN [8] uses a d-dimensional Cartesian coordinate space 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 is stored at the node whose zone contains the point's coordinates. Figure 2(a) shows a 2-dimensional [0,1] x [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.

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, as shown in Figure 2(b). Each node maintains $O(d)$ state, and the lookup cost is $O(dN^{1/d})$.

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 two and assigns one of the halves to the new node. The new node can easily initialize its routing table, since all its neighbors, except n itself, are among 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. To handle node failures, CAN allows the neighbor of a failed node 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 node-reassignment algorithm in the background. This algorithm tries to assign zones that can be merged into a valid zone to the same node, and then combine them.

Summary and Open Questions

The lookup algorithms described here are all currently under development. Their 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 here to help contrast the algorithms and highlight areas for future work.

Distance function. The choice of distance function has implications for other aspects of the algorithms. For example, Kademlia's XOR-based function has the nice property of being unidirectional (for any given point x and distance $d > 0$, there is exactly one point y such that the distance between x and y is d) and symmetric (the distance from x to y is equal to the distance from y to x) [6]. Chord is unidirectional, but not symmetric; Pastry is symmetric but not unidirectional. Because the metric is symmetric, there is no need for a stabilization protocol like Chord; routing tables are refreshed as a side effect of ordinary lookups. Because the metric is unidirectional, Kademlia doesn't need a leaf set like Pastry.

Operation costs. The routing strategies described here have all been analyzed under static conditions. A key area for future analysis 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. A promising approach is based on a notion of the "half-life" of a system [4].

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 Tapestry also guarantee correctness for the difficult case of concurrent joins by nodes with similar IDs, as well as for simultaneous failures. Some research focuses on algorithms that improve efficiency under failures by avoiding timeouts to detect failed nodes [5, 6, 10].

Proximity routing. CAN, Kademlia, Pastry, and Tapestry have heuristics to choose routing-table entries refering 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—a new version uses an algorithm proposed by Karger and Ruhl for proximity routing [3]. 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 effective on the real Internet topology.

Malicious nodes. Pastry uses certificates to prove node identity, allowing strong defenses against malicious participants. The cost, however, is trust in a 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 achieve 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. These DHT algorithms retrieve data based on a unique identifier. In contrast, the widely deployed P2P file-sharing services 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 algorithms have many aspects in common, but comparing them also reveals a number of issues that require further investigation to resolve. They all share the DHT abstraction, and this has been shown to be beneficial in a range of distributed P2P applications. With more work, DHTs might well prove to be a valuable building block for robust, large-scale distributed applications on the Internet. C

References

1. Clarke, I., Sandberg, O., Wiley, B., and Hong, T. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of ICSI Workshop on Design Issues in Anonymity and Unobservability*. Berkeley, California (June 2000); freenet.sourceforge.net.
2. Hildrum, K., Kubiatowicz, J., Rao, S., and Zhao, B. Distributed Object Location in a Dynamic Network. In *Proceedings of 14th ACM Symp. on Parallel Algorithms and Architectures* (SPAA), August 2002.
3. Karger, K., and Ruhl M., Finding nearest neighbors in growth-restricted metrics. In *Proceedings ACM Symp. on the Theory of Computing* (May 2002), 741–750.
4. Liben-Nowell, D., Balakrishnan H., and Karger, D. Analysis of the evolution of peer-to-peer systems. In *Proceedings in ACM Symp. on the Principles of Distributed Computing*. Monterey, CA (July 2002).
5. Malkhi, D., Naor, M., and Ratajczak, D. Viceroy: A scalable and dynamic emulation of the butterfly. In *Proceedings of ACM Principles of Distributed Computing* (PODC) Monterey, CA (July 2002).
6. Maymounkov, P., and Mazieres, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, Springer-Verlag version, Cambridge, MA (Mar. 2002); kademia.scs.cs.nyu.edu.
7. Plaxton, C., Rajaraman, R., and Richa, A. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of ACM Symp. on Parallel Algorithms and Architectures* (SPAA), Newport, Rhode Island (June 1997).
8. Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, San Diego, CA (August 2001).
9. Rowstron, A., and Druschel, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms* (Nov. 2001); www.cs.rice.edu/CS/Systems?Pastry.
10. Saia, J., et al. Dynamically fault-tolerant content addressable networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA (March 2002); oceanstore.cs.berkely.edu.
11. Stoica, I., et al. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of ACM SIGCOMM*, San Diego (August 2001); www.pdos.lcs.mit.edu/chord.

Hari Balakrishnan (hari@lcs.mit.edu) is an associate professor in the Department of Electric Engineering and Computer Science Department (EECS) and a member of the Lab for Computer Science (LCS) at MIT, Cambridge, MA.

M. Frans Kaashoek (kaashoek@lcs.mit.edu) is a professor of computer science and engineering in MIT's EECS and a member of LCS.

David Karger (karger@lcs.mit.edu) is an assistant professor in the EECS Department and a member of LCS.

Robert Morris (rtm@lcs.mit.edu) is an assistant professor in the EECS Department and a member of LCS.

Ion Stoica (istoica@cs.berkeley.edu) is an assistant professor in the EECS Department at the University of California, Berkeley.