Building Peer-to-Peer Systems
With Chord, a Distributed Lookup Service

Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger
Robert Morris, Ion Stoica*, Hari Balakrishnan
MIT Laboratory for Computer Science
{*fdabek, emma, kaashoek, karger, rtm, hari*}*@lcs.mit.edu*
`http://pdos.lcs.mit.edu/chord`

Abstract

We argue that the core problem facing peer-to-peer systems is locating documents in a decentralized network and propose Chord, a distributed lookup primitive. Chord provides an efficient method of locating documents while placing few constraints on the applications that use it. As proof that Chord's functionality is useful in the development of peer-to-peer applications, we outline the implementation of a peer-to-peer file sharing system based on Chord.

1 Introduction

The peer-to-peer architecture offers the promise of harnessing the resources of vast numbers of Internet hosts. The primary challenge facing this architecture, we argue, is efficiently locating information distributed across these hosts in a decentralized way. In this paper we present Chord, a distributed lookup service that is both scalable and decentralized and can be used as the basis for general purpose peer-to-peer systems.

A review of the features included in recent peer-to-peer systems yields a long list. These include redundant storage, permanence, efficient data location, selection of nearby servers, anonymity, search, authentication, and hierarchical naming. Chord does not implement these services directly but rather provides a flexible, high-performance lookup primitive upon which such functionality can be efficiently layered. Our design philosophy is to separate the lookup problem from additional functionality. By layering additional features on top of a core lookup service, we believe overall systems will gain robustness and scalability.

In contrast, when these application-level features are an integral part of the lookup service the cost is often limited scalability and diminished robustness. For example, Freenet [5] [6] is designed to make it hard to detect which hosts store a particular piece of data, but this feature prevents Freenet from guaranteeing the ability to retrieve data.

Chord is designed to offer the functionality necessary to implement general-purpose systems while preserving maximum flexibility. Chord is an efficient distributed lookup system based on consistent hashing [10]. It provides a unique mapping between an identifier space and a set of nodes. A node can be a host or a process identified by an IP address and a port number; each node is associated with a Chord identifer. Chord maps each identifier a to the node with the smallest identifier greater than a. This node is called the *successor* of a.

By using an additional layer that translates high level names into Chord identifiers, Chord may be used as a powerful lookup service. We will outline the design of a distributed hash table (DHASH) layer and of a peer-to-peer storage application based on the Chord primitive. Figure 1 shows the distribution of functionality in the storage application.

Layer	Function
Chord	Maps identifiers to successor nodes
DHASH	Associates values (blocks) with identifiers
Application	Provides a file system interface

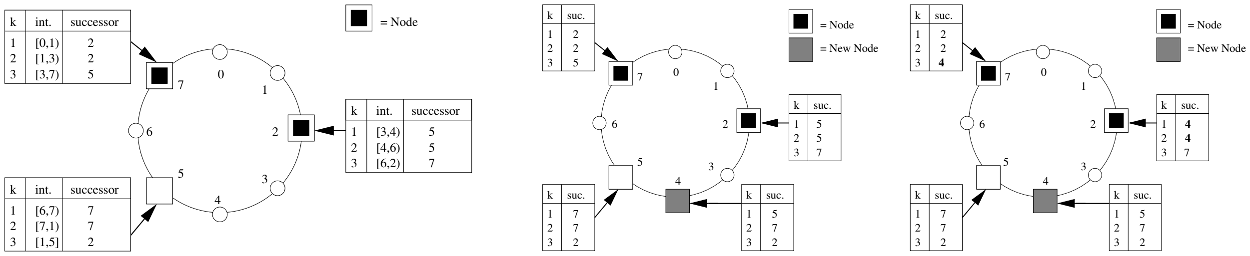
Figure 1: A layered Chord application

Chord is efficient: determining the successor of an identifier requires that $O(\log N)$ messages be exchanged with high probability where N is the number of servers in the Chord network. Adding or removing a server from the network can be accomplished, with high probability, at a cost of $O(\log^2 N)$ messages.

The rest of this position paper outlines the algorithms used to implement the Chord primitive (Section 2), describes how Chord can be used to build peer-to-peer storage systems (Section 3), summarizes the current implementation status of the system (Section 4), identifies some open research problems (Section 5), relates Chord to other work (Section 6), and summarizes our conclusions (Section 7).

* University of California, Berkeley. istoica@cs.berkeley.edu

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.



(a) Find successor (b) Initialize routing tables (c) Update other routing tables

Figure 2: The Chord algorithm in a three-bit identifier space

2 Chord

Chord uses consistent hashing [10] to map nodes onto an m-bit circular identifer space. In particular, each identifier a is mapped to the node with the least identifier greater or equal to a in the circular identifier space. This node is called the *successor* of a.

To implement the *successor* function, all nodes maintain an m-entry routing table called the *finger* table. This table stores information about other nodes in the system; each entry contains a node identifier and its network address (consisting of an IP address and a port number). The k-th entry in the finger table of node r is the smallest node s that is greater than $r + 2^{k-1}$. Node s is also termed the order-k successor of node r. The number of unique entries in the finger table is $O(\log N)$. The finger table can also be thought of in terms of m identifier intervals corresponding to the m entries in the table: the order-k interval of a node r is defined as $((r+2^{k-1}) \bmod 2^m, (r+2^k) \bmod 2^m]$. Figure 2(a) shows a simple example in which m=3 and three nodes 2, 5, and 7 are present. The immediate successor of node 5 is the successor of $(5 + 2^0) \bmod 2^3 = 6$ or node 7.

Each node also maintains a pointer to its immediate predecessor. For symmetry, we also define the corresponding immediate successor (identical to the first entry in the finger table). In total, each node must maintain a finger table entry for up to $O(\log N)$ other nodes; this represents a significant advantage over standard consistent hashing which requires each node to track almost all other nodes.

2.1 Evaluating the successor function

Since each node maintains information about only a small subset of the nodes in the system, evaluating the successor function requires communication between nodes at each step of the protocol. The search for a node moves progressively closer to identifying the successor with each step.

A search for the successor of f initiated at node r begins by determining if f is between r and the immediate successor of r. If so, the search terminates and the successor of r is returned. Otherwise, r forwards the search request to the largest node in its finger table that precedes f; call this node s. The same procedure is repeated by s until the search terminates.

For example, assume that the system is in a stable state (all routing tables contain correct information) and a search is initiated at node 2 of Figure 2(a) for the successor of identifier 6. The largest node with an identifier smaller than 6 is 5. The target of the search, 6, is in the interval defined by 5 and its successor (7); therefore 7 is returned value.

The algorithm is outlined above in *recursive* form: if a search request requires multiple steps to complete, the n^{th} step is initiated by the $(n-1)^{th}$ node on behalf of the initiator. The successor function may also be implemented *iteratively*. In an iterative implementation, the initiating node is responsible for making requests for finger table information at each stage of the protocol. Both implementation styles offer advantages: an iterative approach is easier to implement and relies less on intermediary nodes, while the recursive approach lends itself more naturally to caching and server selection (described in Section 3).

2.2 Node insertion

When a new node r joins the network it must initialize its finger table; existing nodes must also update their tables to reflect the existence of r (see Figure 2(b) and Figure 2(c)).

If the system is in a stable state, a new node r can initialize its finger table by querying an existing node for the respective successors of the lower endpoints of the k intervals in r's table. Although we omit the details here, nodes whose routing information is invalidated by r's addition can be determined using r's finger table and by following predecessor pointers: these nodes are instructed by r to update their tables.

2.3 Additional algorithm details

Several additional details of the Chord protocol are merely mentioned here in the interest of brevity; a complete description of the Chord primitive is given by Stoica et al. [17]. Removing a node from the network involves a sim-

```
void event_register ((fn)(int))
ID next_hop (ID j, ID k)
```

Figure 3: Exposing Chord layer information. The `event_register` function arranges for `fn` to be called when a node with an ID near the registrant's joins or leaves the network. `next_hop` performs one step of the evaluation of the successor function and returns the intermediate result (a finger table entry).

ilar series of steps as adding a node. Parallel joins, parallel exits, and failures are handled by maintaining the invariant that all nodes are aware of their immediate successor and predecessor, and by allowing the remaining entries of nodes' finger tables to converge to the stable state over time. Handling failures also requires that nodes store k successors in addition to the immediate successor.

2.4 The chord library API

The Chord library is intended to be used in a layered design where it provides the base location functionality. Two design principles facilitate the the use of Chord in a layered architecture: minimum functionality and exposed information. By minimizing the amount of functionality embedded in Chord, we minimize the constraints we place on higher levels which depend on Chord.

In our initial experiments with systems based on Chord, we found that larger systems were constrained not because Chord provides an inflexible feature set, but because higher layers desired access to the internal state of Chord during its computation.

To provide this access while still preserving the abstraction barrier we allow layers to register callback functions for events they are interested in (see Figure 3) and to evaluate the successor function one step at a time. `next_hop(j, k)` sends a message to node `j` asking `j` for the smallest entry in its finger table greater than `k`. This allows callers to control the step-by-step execution of the Chord lookup algorithm.

For example, the `DHASH` layer (described in section 3.1) uses the callback interface to move values when nodes join or leave the system. `DHASH` also evaluates the successor function step by step to perform caching on search paths.

3 Building on Chord

To illustrate the usefulness of the Chord API we will outline the design of layers that could be built on the basic Chord primitive. These layers would be useful in a larger peer-topeer file sharing application. This application should allow a group of cooperating users to share their network and disk resources. Possible users of the application might be a group of open source developers who wish to make a software distribution available, but individually do not have network resources to meet demand.

```
err_t insert(void *key, void *value)
void * lookup(void *key)
```

Figure 4: The DHASH API (a) Inserts value under key (b) returns value associated with key or NULL if key does not exist

3.1 Distributed hash service

Chord is not a storage system: it associates keys with nodes rather than with values. A useful initial extension to this system is a distributed hash table (DHASH). The API for this layer is shown in Figure 4.

`DHASH::insert` can be implemented by hashing `key` to produce a 160-bit Chord identifier k, and storing `value` at the successor of k. A `DHASH::lookup` request is handled analogously: `key` is hashed to form k and the successor of k is queried for the value associated with `key`. The transfer of value data to and from nodes is accomplished by an additional RPC interface which is separate from that exported by Chord.

Values introduce a complication: when nodes leave or join the system, the successor node of a given key may change. To preserve the invariant that values are stored at the successor of their associated keys, `DHASH` monitors the arrival and departure of nodes using the callback interface provided by Chord and moves values appropriately. For example, if the value associated with key 7 is stored on node 10 and node 9 joins the system, that value will be transferred to node 9.

Because it is based on Chord, `DHASH` inherits Chord's desirable properties: performing a lookup operation requires $O(\log N)$ RPCs to be issued and does not require any centralized control. The `DHASH` layer imposes an additional cost of transferring $O(\frac{1}{N})$ of the keys in the system each time a node joins or leaves the system.

3.2 Achieving reliability

The `DHASH` layer can also exploit the properties of Chord to achieve greater reliability and performance. To ensure that lookup operations succeed in the face of unexpected node failures, `DHASH` stores the value associated with a given key not only at the immediate successor of that key, but also at the next r successors. The parameter r may be varied to achieve the desired level of redundant storage.

The tight coupling between `DHASH`'s approach to replication and Chord's (both use knowledge of a node's immediate successors) is typical of the interaction we hope to see between Chord and higher layers.

3.3 Improving performance

To improve `DHASH` lookup performance, we exploit a property of the Chord lookup algorithm: the paths that searches for a given successor (from different initiating nodes) take through the Chord ring are likely to intersect. These intersections are more likely to occur near the target of the search where each step of the search algorithm makes a smaller 'hop' through the identifier space and provide an opportunity to cache data. On every successful lookup operation of a pair (k, v), the target value, v, is cached at each node in the path of nodes traversed to determine the successor of k (this path is returned by Chord's successor function).

Subsequent lookup operations evaluate the successor function step by step using the provided `next_hop` method and query each intermediate node for v; the search is terminated early if one of these nodes is able to return a previously cached v.

As a result, values are "smeared" around the Chord ring near corresponding successor nodes. Because the act of retrieving a document caches it, popular documents are cached more widely than unpopular documents; this is a desirable side-effect of the cache design. Caching reduces the path length required to fetch a value and therefore the number of messages per operation: such a reduction is important given that we expect that latency of communication between nodes to be a serious performance bottleneck facing this system.

3.4 Denial of service

The highly distributed nature of Chord helps it resist many but not all denial of service attacks. For instance, Chord is resistant to attacks that take out some network links since nodes nearby in identifier space are unlikely to have any network locality. Additional steps are taken to preclude other attacks.

A Chord-based storage system could be attacked by inserting such a large volume of useless data into the system that legitimate documents are flushed from storage. By observing that the density of nodes nearby any given node provides an estimate of the number of nodes in the system we can partially defend against this attack by limiting the number of blocks any one node can store in the system. We make a local decision to fix a block quota based on the number of nodes in the system, effectively enforcing a fixed quota for each user on the whole system.

Nodes that could pick their own identifiers could effectively delete a piece of data from the system by positioning themselves as the data's successor and then failing to store it when asked to. This attack can be prevented by requiring that node identifiers correspond to a hash of a node's IP address, a fact which can be verified by other nodes in the system.

Malicious nodes could fail to execute the Chord protocol properly resulting in arbitrarily incorrect behavior. A single misbehaving node can be detected by verifying its responses with those of other, presumably cooperative, nodes. For instance, if a node l reports that its successor is s, we can query s for its predecessor which should be l. A group of such nodes could cooperate to make a collection of nodes appear to be a self-consistent Chord network while excluding legitimate nodes. We have no decentralized solution to this problem and rely instead on the legitimacy of the initial 'bootstrap' node to avoid this attack.

3.5 Designing a storage system: balancing load

In using Chord as the core of a peer-to-peer storage system we are faced with the problem of efficiently distributing load among nodes despite wide variations in the popularity of documents. In building this system we must consider how to map documents to nodes and at what granularity to store documents.

One might consider using DHASH directly as a peer-to-peer storage system. In this design, the contents of a document are directly inserted into the DHASH system keyed by the hash of either the contents of the document or, perhaps, a human readable name. If one document becomes highly popular, however, the burden of delivering that document will not be distributed. The caching scheme described in Section 3.3 helps for small documents, but is not practical for very large documents.

An alternate approach uses DHASH as a layer of indirection: DHASH maps document identifiers to a list of IP addresses where that document was available. In this design DHASH functions analogously to the DNS system but does not depend on a special set of root servers as DNS does. Once an IP address is selected, documents are retrieved using some other transfer protocol (HTTP, SSL, SFS etc.).

Maintaining a dynamically updated list of potential servers for any document solves the problem of popular documents by distributing load among all of the servers in the list. However, this design requires that optimizations such as caching and redundant storage be implemented twice: once in the Chord stack and again in the transfer protocol. We desire a tighter coupling between the solution to the popular document problem and mechanisms of the Chord protocol.

This coupling can be achieved by using Chord to map pieces of documents (blocks), rather than whole documents, to servers. In this scheme, files are broken into blocks and each block is inserted into the `DHASH` layer using the cryptographic hash of the block's contents as a key. A piece of meta-data, equivalent to an inode in a traditional file system, is also inserted into the system to provide a single name for the file. The equivalence to a file system can be extending to include directories as well; in our prototype implementation, names map to a directory of documents which is mapped into the user's local namespace when accessed.

This approach aggressively spreads a single large document across many servers, thus distributing the load of serving it. It also inherits the reliability and performance enhancements of the DHASH layer with little or no additional effort. One might note that documents smaller than the block size are still served by a single node: we count on our caching scheme to distribute these documents and the load of serving them if they become popular.

The major drawback of this scheme derives from the same property that made it desirable: because we spread a single document across many servers, for each document we fetch we must pay the cost of several DHASH lookups (and thus several evaluations of the successor function). A naive implementation might require $\frac{S \times L \times \log N}{B}$ seconds to fetch an S byte document where N is the number of servers in the network, B is the block size and L is the average latency of the network. We hope to hide most of this latency through aggressive prefetching of data and by selecting a server from the redundant set which is near (in the network) the requesting node.

3.6 Authenticity

A Chord-based file system could achieve authenticity guarantees through the mechanisms of the SFS read-only server [9]. In SFSRO, file system blocks are named by the cryptographic hash of their contents, an inherently unforgeable identifier. To name file systems we adopt self-certifying pathnames [14]: The block containing the root inode of a file system is named by the public key of the publisher and signed by that public key. The DHASH layer can verify that the root inode is correctly signed by the key under which it is inserted. This prevents unauthorized updates to a file system. Naming file systems by public key does not produce easily human readable file names; this is not a serious shortcoming, however, in a hypertext environment, or one that is indexed or provides symbolic links.

4 Status

The system described is under development. The Chord protocol has been designed, implemented, and tested[1]. Results of testing with up to 1,000 nodes on the Berkeley Millennium Cluster demonstrate that Chord's performance scales well with the size of the system. We have also implemented the DHASH layer and a file system; in the same testing environment and on a geographically diverse network both demonstrated good load balancing properties.

5 Open problems

A number of open problems face applications built on the Chord framework.

Our design deliberately separates questions of anonymity and deniability from the location primitive. These properties are difficult to add to the Chord system given the strong mapping between a document and the node which is responsible for serving that document. We speculate than overlaying a mix-network [4] on Chord might allow for anonymous publishing and reading.

Collecting an index of all documents stored in Chord is a straightforward operation: an indexer might visit every node in the Chord system by following successor pointers. Storing an index and servicing queries without resort to a central authority remains an open question, however. Alternatively we could provide a Chord to WWW gateway and rely on existing WWW indexing services.

Directing requests to servers nearby in the network topology is important to reducing the latency of requests. To do so requires measuring the performance of servers in the system. However, because Chord aggressively distributes documents to unrelated servers, in a large network we are not likely to visit the same server multiple times; this makes maintaining server performance metrics difficult.

6 Related work

There has been previous work in the area of decentralized location systems. Chord is based on consistent hashing [10]; its routing information may be thought of as a one-dimensional analogue of the GRID [12] location system. OceanStore [11] uses a distributed data location system described by Plaxton et al. [7], which is more complicated than Chord but offers proximity guarantees. CAN uses a d-dimensional Cartesian coordinate space to implement a distributed hash table data structure [16]. CAN operations are easy to implement, but an aditional maintenance protocol is required to periodically remap the identifier space onto nodes. The Chord algorithm is also very similar to the location algorithm in PAST [15].

Anonymous storage systems such as Freenet [5], Publius [13] and the Free Haven Project [8] use encryption, probabilistic routing, or secret-sharing schemes to guarantee clients and publishers anonymity. This anonymity guarantee often leads to design compromises that limit reliability and performance. Chord separates problems like these from the design of routing and file transfer protocols.

Napster [2], Gnutella [1], and Ohaha [3] provide a non-anonymous file sharing service similar to that of the sharing application presented here. Chord's location algorithm is more efficient than Gnutella's broadcast based routing; the decentralized nature of Chord eliminates a single point of failure present in Napster. The Ohaha system [3] uses a consistent hashing-like algorithm for ID mapping, and a Freenet-style method of document retrieval; it shares some of the weaknesses of Freenet.

[1]The delete operation has not been implemented yet

7 Conclusions

The performance and reliability of existing peer-to-peer systems have been limited by inflexible architectures that attempt to find one solution for many problems. By using the Chord primitive to separate the problem of location from the problems of data distribution, authentication and anonymity, peer-to-peer systems are able to decide where to compromise and as a result offer better performance, reliability and authenticity.

References

[1] Gnutella website. http://gnutella.wego.com.

[2] Napster. http://www.napster.com.

[3] Ohaha. http://www.ohaha.com/design.html.

[4] David Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. *Communications of the A.C.M.*, 24(2):84–88, 1981.

[5] Ian Clarke. A distributed decentralised information storage and retrieval system. Master's thesis, University of Edinburgh, 1999.

[6] Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, California, June 2000. http://freenet.sourceforge.net.

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

[8] Roger Dingledine, David Molnar, and Michael J. Freedman. The Free Haven project: Distributed anonymous storage service. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[9] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 181–196, San Diego, California, October 2000.

[10] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[11] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Boston, MA, November 2000.

[12] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 120–130, Boston, Massachusetts, August 2000.

[13] Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[14] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 124–139, Kiawah Island, South Carolina, December 1999.

[15] Antony Rowstron Peter Druschel. Past: Persistent and anonymous storage in a peer-to-peer networking environment. In *Proceedings of the 8th Conference on Hot Topics in Operating Systems (HotOS 2001)*, May 2001.

[16] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM 2001*, August 2001.

[17] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM 2001*, August 2001. An early version appeared as LCS TR-819 available at `http://www.pdos.lcs.mit.edu/chord/papers`.