A Common Protocol for Implementing Various DHT Algorithms

Salman Baset Henning Schulzrinne Eunsoo Shim

Department of Computer Science
Columbia University
New York, NY 10027
{salman,hgs}@cs.columbia.edu

Panasonic Princeton Laboratory
Two Research Way, 3rd Floor
Princeton, NJ 08540
eunsoo@research.panasonic.com

Abstract

This document defines DHT-independent and DHT-dependent features of DHT algorithms and presents a comparison of Chord, Pastry and Kademlia. It then describes key DHT operations and their information requirements.

1 Introduction

Over the last few years a number of distributed hash table (DHT) algorithms [1, 2, 3, 4, 5] have been proposed. These DHTs are based on the idea of consistent hashing [6] and they share a fundamental function, namely routing a message to a node responsible for an identifier (key) in $O(\log_{2^b} N)$ steps using a certain routing metric where N is the number of nodes in the system and b is the base of the logarithm. Identifiers can be pictured to be arranged in a circle in Chord [1], Kademlia [4] and Pastry [2] and a routing metric determines if the message can traverse only in one direction ([anti-]clockwise) or both directions on the identifier circle. However, independent of the routing metric and despite the fact that the author of these DHT algorithms have given different names to the routing messages and tables, the basic routing concept of $O(\log_2 N)$ operations is the same across DHTs.

In this paper, we want to understand if it is possible to exploit the commonalities in the DHT algorithms such as Chord [1], Pastry [2], Tapestry [3] and Kademlia [4] to define a protocol by which any of these algorithms can be implemented. We have chosen Chord, Pastry and Kademlia because either they have been extensively studied (Chord and Pastry) or they have been used in a well-deployed application (Kademlia in eDonkey [7]). We envision that the protocol should not contain any algorithm-specific details and possibly have an extension mechanism to incorporate an algorithm-specific feature. The goal is to minimize the possibility of extensions that may unnecessarily complicate the protocol.

We first define the terminology and metrics used in our comparison of DHTs in section 2 and 3 and then give a brief description of Chord, Pastry and Kademlia in section 4. The authors of these algorithms have proposed a number of heuristics to improve the lookup speed and performance such as proximity neighbor selection (PNS) [2] which should not be considered part of the core algorithm. We carefully separate DHT-independent heuristics from DHT-specific details and try to expose the commonality in these algorithms in section 5. Using this commonality, we then define algorithm-independent functions such as join, leave, keep-alive, insert and lookup and discuss protocol semantics and information requirements for these functions in section 6.

2 Terminology

Some of the terminology has been borrowed from the P2P terminology draft [8].

P2PSIP Overlay Peer (or Peer). A P2PSIP peer is a node participating in a P2PSIP overlay that provides storage and routing services to other nodes in the P2P overlay and is capable of performing operations such as joining and leaving the overlay and routing requests within the overlay. We use the term node and peer interchangeably.

P2PSIP Client (or Client). A P2PSIP client is a node participating in a P2PSIP overlay that provides neither routing nor route storage and retrieval functions to that P2PSIP Overlay.

Key A key is an information that uniquely identifies a datum stored in the P2PSIP overlay. A key refers to Peer-IDs and datum keys. In the DHT approach, this is a numeric value in the hash space.

P2PSIP Peer-ID or ID. A Peer-ID is an information (key) that uniquely identifies a peer within a given P2PSIP overlay.

Routing table. A routing table is used by a node to map a key to a peer responsible for that key. It contains a list of online peers and their IDs. Each row of the routing table can be thought of as having an ID (thereafter called row-ID) which is from the same key space as the peer-ID. The row-ID i is exponentially closer to the node-ID than the row-ID $(i + 1)$ and is computed by applying a DHT-algorithm specific function. The i^{th} row can only store the IP address of one or peers whose peer-ID lie between i^{th} and $(i+1)^th$ row-IDs. Simplistically, the number of entries in the routing table is logarithmic to the number of nodes in the DHT network.

A *Chord peer* computes its i^{th} row-ID by performing the following modulo arithmetic:

$$(\text{Peer-ID} + 2^{i-1})\ \%\ 2^M \text{ where M is the key length and } i \text{ is between 1 and M.}$$

A *Pastry peer* with a base b computes its i^{th} routing table key using the following function:

$$prefix(M - i) \text{ where M is the length of the key.}$$

Each row in the routing table of a Pastry peer contains $2^b - 1$ entries that whose ID match the peer-ID in $(M - i)$ digits but the $(M - i) + 1^{th}$ digit has one of the $2^b - 1$ possible values other than the $(M - i) + 1^{th}$ digit present in the peer-ID.

Routing table row interval (or Row-ID Interval). The routing table row interval for row i is defined as the number of keys that lie between row-ID i and row-ID $i + 1$ according to a DHT-algorithm specific function.

3 Description of DHT Specific Metrics

Below, we give an explanation of metrics which we believe are significant in our comparison of DHTs.

3.1 Distance Function

Any peer which receives a query for a key k must forward it to a peer whose ID is ‘closer’ to k than its own ID. This rule guarantees that the query eventually arrives at the peer responsible for the key. The closeness does not represent the way a routing table is filled but rather how a node in the routing table is selected to route the query towards its destination. Closeness is defined as follows in Chord, Pastry and Kademlia [9]:

Chord. Numeric difference between two keys. Specifically, for two keys a and b:
$(b - a) \, mod \, 2^M$ where M is the length of the key produced by a hash function.

Pastry. Inverse of the number of common prefix-bits between two keys.

Kademlia. Bit-wise exclusive-or (XOR) of the two keys. Specifically for two keys a and b:
$a \oplus b$

Pastry uses numerical difference when prefix-matching does not match any additional bits and the peers which are closer by prefix-matching metric may not be closer by the numerical difference metric.

3.2 Routing Table Rigidity

There are two ways in which a peer can select a node for its i^{th} routing table row. It can either select a node for row i such that the peer-ID of the node is immediately before or after (in the DHT sense) the row-ID i among all online nodes or it can select a node whose ID lies within the i^{th} row-ID interval (see terminology section for a definition of row-ID interval). For its i^{th} row, and amongst all online nodes, Chord selects a node with a peer-ID which is immediately after row-ID i while Pastry and Kademlia can pickup any node having an ID that lies within the i^{th} row-ID interval. The effect of this is that Pastry and Kademlia have more flexibility in selecting peers for their routing table while Chord has a rather strict criteria. It is possible to loosen the selection criteria in Chord by selecting any node in the interval without violating the $\log_2 N$ bound.

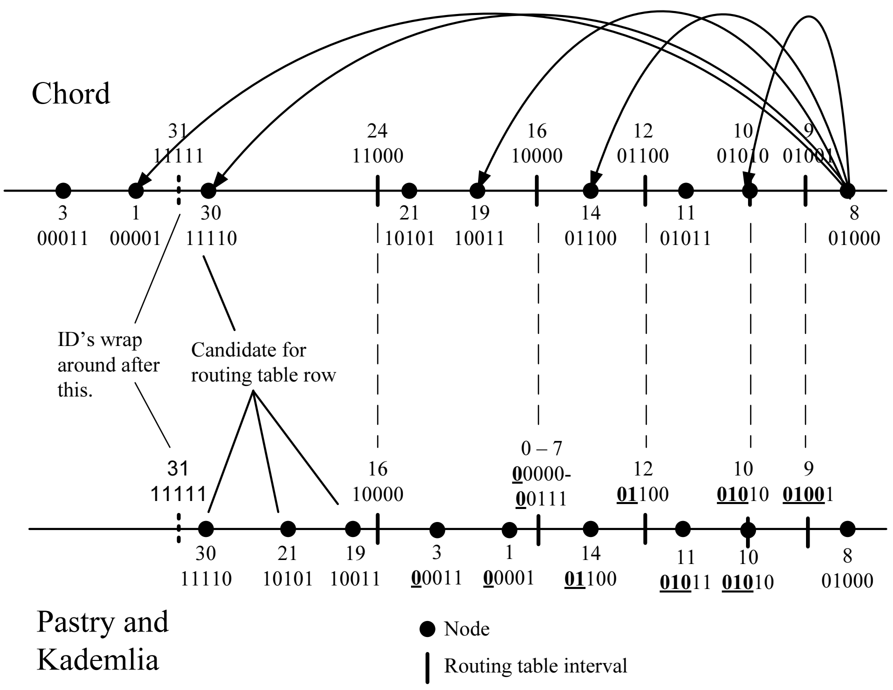
Moreover, in Chord, a lookup query will never overshoot the key, i.e., it will never be sent to a node whose ID is greater than the key being queried. Since Pastry and Kademlia can pickup any node in the interval, a lookup query can possibly overshoot the key. Figure 3.2 shows how a peer having the same ID selects routing table entries in Chord, Kademlia and Pastry.

3.3 Learning from Lookup Queries

The mechanism for selecting a node for a routing table row directly impacts whether a peer can update its routing table from a lookup query it receives. If, for its i^{th} routing table row, a peer always selects a node amongst all online nodes whose ID is immediately before or after the row-ID i, then the number of such peers is only one. However, choosing any peer whose ID lies within the i^{th} row-ID interval provides more flexibility as the number of candidate nodes increase from one to the number of online peers in the interval. A node which intends to update its routing table from the lookup queries it receives has a better chance of doing so.

3.4 Sequential vs Parallel Lookups

If a querying node's routing table row contains IP address of two or more DHT nodes, then it may send a lookup query to all of them. The reason any node will send parallel lookup queries is because the routing table peers may not have been refreshed for sometime and thus may not be online. If all nodes in a DHT frequently refresh their routing table, then there may not be a need to send parallel queries even in a reasonably high churn environment. A node can reduce the number of parallel queries it sends by by reducing the interval between periodic keep-alives to ensure the liveness of its routing table entries. Thus, there is a tradeoff between sending keep-alives to routing table peers, and sending parallel lookup queries.



Chord routing table for node 8	
Row-ID	Candidate nodes
9	10
10	10
12	14
16	19
24	30

Pastry routing table for node 8 (01000)	
Row-ID	Candidate nodes
9 (0100X)	10
10 (010XX)	10
12 (01XXX)	14
0 (0XXXX)	1, 3
16 (XXXX)	19, 21, 31

Figure 1: The figure shows the candidate routing table nodes for a peer with an ID *8* in Chord, Kademlia and Pastry. The bold and underlined bits show the number of prefix bits matching with the node ID. The 'X' in the Pastry routing table represent the first digit in which the row-ID differs from the peer-ID. All digits following this digit are also labeled as 'X'.

3.5 Iterative vs Recursive Lookups

In an iterative lookup, the querying peer sends a query to a node in its routing table which replies with the IP address of the next hop if it is not responsible for the key. The querying peer then sends the query to this node. In a recursive lookup, the querying peer sends a query to a node in its routing table, which after receiving the lookup query applies the appropriate DHT metric to determine the next hop peer, and forwards the query to this peer without replying to the querying peer. Once the node responsible for the key is found, it sends a message directly to the querying peer instead of sending it back to the peer which forwarded the query. Rhea [10] explains the differences between iterative vs recursive lookups.

The recursive lookup can possibly cause a mis-configured or misbehaving node to start a flood of queries in a DHT. On the other hand, recursive lookup as described above can provide lower latencies than iterative lookup since the node responsible for the key sends the key/value pair directly to the querying peer instead of the peer which forwarded this query. This can cause security issues as the querying peer now has to process responses from peers to whom

Table 1: DHT independent details

	Key-length	Recursive/ Iterative	Sequential/ Parallel	Routing table name	Neighbor Nodes
Chord	160	Both	Sequential	Finger table	Successor list
Pastry	128	Recursive	Sequential	Routing table	Leaf-set
Kademlia	160	Iterative	Parallel	Routing table	None

Table 2: DHT specific details

	Routing Data Structure	Routing table row selection	Symmetric	Learning	Overshooting
Chord	Skip-list	Immediately succeed the interval	No	No	No
Pastry	Tree-like	Any node in the interval	Yes	Yes	Yes
Kademlia	Tree-like	Any node in the interval	Yes	Yes	Maybe

it never sent a request.

4 Chord, Pastry and Kademlia

In this section, we try to expose commonalities in Chord, Pastry and Kademlia. These algorithms are based on the idea of consistent hashing [6], i.e., keys are mapped onto nodes by a hash function that can be resolved by any node in the system via queries to other nodes and the arrival or departure of a node does not require all keys to be rehashed. We start by comparing DHT independent details of these algorithms as defined by their authors in Table 1 and then algorithm-specific details in Table 2 and then give a brief description of Chord, Pastry and Kademlia.

4.1 Chord

The identifiers or keys in Chord can be pictured to be arranged on a circle. Each node in Chord maintains two data structures, a *successor list* which is the list of peers immediately succeeding the node key and a *finger table*. A finger table is a routing table which contains the IP address of peers halfway around the ID space from the node, a quarter-of-the-way, an eighth-of-the-way and so forth in a data structure that resembles a skiplist [11]. A node forwards a query for a key k to a node in its finger (routing) table with the highest ID **not** exceeding k. The skiplist structure ensures that a key can be found in $O(\log_2 N)$ steps where N is the number of nodes in the system. This structure can be extended to use a logarithm base higher than two.

To join a Chord ring, a node contacts any peer in the Chord network and requests it to lookup its ID. It then inserts itself at the appropriate position in the Chord network. The predecessors of the newly joined node must update their successor lists. The newly joined node should also update its finger table. Successor list is the only requirement for correctness while finger table is used to speedup the lookups.

To guard against node failures, Chord sends keep-alives to its successors and finger table entries and continuously repairs them. The routing table size is $\log_2 N$. For an arbitrary logarithm base b, the routing table size is $\log_{2^b} N \times (2^b - 1)$.

Chord suggests two ways for key/data replication. In the first method, an application replicates data by storing it under two different Chord keys derived from the data's key. Alternatively, a Chord node can replicate key/value pair on each of its r successors.

4.2 Pastry

Like Chord, the identifiers or keys in Pastry can be pictured to be arranged on a circle; however, the routing is done in a tree-based (prefix-matching) fashion. Each node in Pastry contains two data structures, a *leaf-set* and a *routing table.* The leaf-set L contains $|L|/2$ closest nodes with numerically smaller identifiers than the node n and $|L|/2$ closest nodes with numerically larger identifiers than n and is conceptually similar to Chord successor list [9]. The routing table contains the IP address of nodes with no prefix match, b bits prefix match, 2b prefix match and so on where b is 1, 2, 3,...,M (M is the length of the key returned by the hash function). The maximum size of the routing table is $\log_{2^b} N$ x 2^b where N is the number of nodes in the system. At each step, a node n tries to route the message to a node that has a longest sharing prefix than the node n with the sought key. If there is no such node, the node n routes the message to a node whose shared prefix is at least as long as n and whose ID is numerically closer to the key. The expected number of hops is at most $\lceil \log_{2^b} \mathrm{N} \rceil$.

To join the Pastry network, a node contacts any node in the Pastry network and builds routing tables and leaf sets by obtaining information from the nodes along the path from bootstrapping node and the node closest in ID space to itself. When a node gracefully leaves the network, the leaf-sets of its neighbors are immediately updated. The routing table information is corrected only on demand.

The routing table of a Pastry node is initialized such that amongst all candidate nodes for a row i sharing a common prefix p_i with the peer-ID, select a node which has the least network latency. This technique is commonly known as proximity neighbor selection (PNS). Pastry performs recursive lookups. However, PNS and recursive lookups are orthogonal to the Pastry operation.

Pastry replicates data by storing the key/value pair on k nodes with the numerically closest nodeIds to a key [2]. This method is conceptually similar to Chord's replication of key/value pairs on its r successors.

4.3 Kademlia

Like Chord and Pastry, the identifiers in Kademlia can be pictured to be arranged on a circle; however the routing is done in a tree-based (prefix-matching) fashion. Each node in Kademlia contains a *routing table.* Kademlia contains only one data structure i.e. the routing table. Unlike Chord and Pastry, there are no successor lists or leaf sets. Rather, the first entry in the routing table serves as the immediate neighbor.

Kademlia uses XOR metric to compute the distance between two identifiers, i.e. $d(x, y) = x \oplus y$. XOR metric is non-Euclidean and it offers the triangle property: $d(x, y) + d(y, z) \geq d(x, z)$. Essentially, XOR metric is a prefix matching algorithm which tries to route a message to a node with the longest matching prefix and the smallest XOR value for non-prefix bits.

Kademlia maintains up to k entries for a routing table row and allows parallel lookups to all nodes in a row. However, this is not really a Kademlia specific feature and other DHT algorithms can implement it by maintaining multiple entries for the same routing table row. A Chord node with a base higher than two contains more than one entry per row.

Table 3: DHT specific RPC

	Keep-alive	Lookup	Store	Join	Updating routing table	Updating neighbor nodes
Chord	fix_fingers()	find_successor()	N/A	join()	fix_fingers()	stabilize()
Pastry	N/A	route(msg,key)	N/A	Side-effect of lookups	On demand	N/A
Kademlia	PING	FIND_NODE, FIND_VALUE, lookup	STORE	N/A	N/A	N/A

The routing table size is $O(\log_2 N)$. The lookup speed can be increased by considering IDs b bits at a time instead of one bit at a time which implies increasing the routing table size. By increasing the routing table size to $2^b log_{2^b} N \times k$ entries, the number of lookup hops can be reduced to $log_{2^b} N$.

Kademlia replicates data by finding k closest nodes to a key and storing the key/value pair on them. The Kademlia paper suggests a value of 20 for k.

5 DHT Commonalities

Table 1 and table 2 list the DHT-independent and DHT-specific aspects of Chord, Pastry and Kademlia. From the above discussion, we deduce following commonalities between Chord, Pastry and Kademlia.

- The time to detect whether a routing entry node has failed is independent of the DHT algorithm being used.
- The flexibility in selecting a node for a routing table row impacts whether a routing table may be updated with information from passing lookup queries.
- Lookup can be performed either iteratively or recursively. Lookup messages can be forwarded either sequentially or parallel.
- It is possible to define replication strategies independent of the underling DHT algorithms.
- The choice of hash function and the length of the key are independent of the routing algorithm.
- Each peer has knowledge about some neighbor nodes.

6 DHT Protocol Operations and their Semantics

In this section, we define and describe DHT operations and information requirements for each operation. But first, we give a brief description of related work.

6.1 Related Work

Dabek et al. [12] defined a key-based API (KBR) which can be used to implement a DHT-level API. They define a RPC **void route(key(K), msg (M), nodehandle (hint))** which forwards a message, M, towards the root of the key K. The optional *hint* specifies a node that should be used as a first hop in routing the message. The *put*() and *get*() DHT operations may be implemented as follows:

- **route(key,[PUT,value,S],NULL)** The *put* operation routes a *PUT* message containing *value* and the local node's handle, S, to the root of the key.
- **route(key,[GET,S],NULL)** The *get* operation routes a *GET* message to the root of the key which returns the value and its own handle in a single hop using **route(NULL,[value,R],S)**.

To replicate a newly received key (k) r times, the peer issues a local RPC **replicaSet(S,r)** and sends a copy of the key to each returned node. The operation implicitly makes the root of the key and not the publisher responsible for replication.

Singh and Schulzrinne [13] defined a XML-RPC based API for DHTs. Their approach is based on OpenDHT [14] and they define a data interface with and without authentication, which allows inserting, retrieving and removing data on a DHT (put, get), and a service interface, which allows a node to join a DHT for a service and another node to lookup for a service node (join, lookup, leave).

We define six DHT operations (API) namely join, leave, insert (put), lookup (get), remove, keep-alive and replicate which a node (peer) participating in a DHT may initiate. A node (client) which does not participate in a DHT network requests a peer in the DHT network to perform these operations on its behalf and thus client-to-peer API is independent of the DHT algorithm being used. The peer-to-peer API can also be independent of the DHT algorithm being used because determination of the next hop is done locally at a peer by applying an algorithm-specific metric.

6.2 Join

A node initiates a *join* operation to a peer already in the DHT to insert itself in the DHT network. The mechanism to discover a peer already in the DHT is independent of any particular DHT being used. The joining node and its neighbors must update their neighbors accordingly.

A joining node may want to build its routing table by getting a full or partial copy of its neighbors or any appropriate node's routing table. It will also need to obtain key/value pairs it will be responsible for.

A join operation initiated by a P2PSIP client does not change the geometry of the DHT network. The operation is conceptually similar to insert(put).

Following is the list of information that will be exchanged between the newly joining node and existing peers.

- [s][1] An overlay ID.
- [s] Peer-ID of the joining node.
- [s] Contact information or IP address of the joining node.
- [s] Indication whether this peer should be inserted in the p2p network thereby changing the geometry or merely stored on an existing peer. This field accommodates overlay peers and clients as defined in [8].
- [r][2] Full or partial routing table of an existing node(s).
- [r] List of immediate neighbors.

[1] **[s]**: a peer sends this information

[2] **[r]**: a peer receives this information perhaps in response to the message it sent

6.3 Leave

A node initiates a *leave* operation to gracefully inform its neighbors about its departure. The neighbors must update their neighbor pointers and take over the keys the leaving node is responsible for.

- [s] The departing node's key.
- [s] List of key/value pairs to be transferred.
- [r] Acknowledgement that the node has been removed from the DHT.

6.4 Insert (put)

A node (overlay client or peer) initiates an *insert* operation to a peer already in the DHT to insert a key/value pair. The insertion involves locating the node responsible for key using the *lookup* operation and then inserting either a reference to the key/value pair publisher or the key/value pair itself. The *insert* operation is different from the *join* operation in the sense that it does not change the DHT geometry. The insert operation can also be used to update the value for an already inserted key.

- [s] Key for the object(value) to be inserted.
- [s] Data item for the key (or value). A sender may choose to only send the value of the key in the *insert* operation after the node responsible for the key has been discovered.
- [s] A flag indicating whether the lookup should be performed recursively or iteratively.
- [s] Publisher of the key. Multiple publishers can publish data under the same key and a node storing a key/value pair uses this field to differentiate among the publishers.
- [s] Key/value lifetime. The time until an online peer must keep the key/value pair. The publisher of the key/value pair must refresh it before the expiration of this time.

6.5 Lookup (get)

A node initiates a *lookup* operation to retrieve a key/value pair from the DHT network. It locally applies DHT routing metric (Chord, Pastry or Kademlia) on its routing table to determine the peer to which it should route the message. The peer responsible for the key/value pair (root of the key) sends it directly back to the querying node. The value can be an IP address, a file or a complex record.

The lookup message can be routed sequentially or in parallel. The lookup message can also be routed iteratively or recursively. A node routing a recursive query may add its own key and IP address information in the lookup message before forwarding it to the next hop.

Following is the list of information exchanged between the querier, forwarding peers and the peer holding the key/value pair.

- [s] Key to lookup.
- [s] A flag indicating whether the lookup should be performed recursively or iteratively.
- [s] Publisher of the key. A non-empty value means that a node is interested in the value inserted by a certain publisher.
- [a][3] Forwarding peer's key and IP address. A node in the path of a lookup query may add its own ID and IP address to the lookup query before recursively forwarding it. This information can be used by the querying peer to possibly update its routing tables.

[3] **[a]**: a peer appends information to the message passing through it

- [r] Data item for the key or an indication that key cannot be found. If the lookup is iterative, this value also indicates whether the lookup is complete or whether the node should send the query to the next-hop peer returned in this message.

6.6 Remove

Even though each stored key/value pair has an associated lifetime and thus will expire unless refreshed by the publishing node in time, sometimes the publishing node may want to remove the key/value pair from the DHT before lifetime expiration. In this case, the publishing node initiates the remove operation.

- [s] Publishing node's key.
- [s] Key for the key/value pair to be removed.
- [r] Acknowledgment that the key has been removed.

6.7 Keep-alive

A peer initiates a *keep-alive* operation to send keep-alive message to its neighbors and routing table entries. The two immediate neighbors do not need to send a periodic keep-alive message to each other. The peers can use various heuristics for keep-alive timer such as randomly sending a keep-alive within an interval.

If a neighbor fails, a peer has to immediately find a new neighbor to ensure lookup correctness. If a routing entry fails, a node may choose to repair it immediately or defer till a lookup request arrives.

- [s] Sending node's key.
- [s] Keep-alive timer expiration.

6.8 Replicate

In order to ensure that a key is not lost when the node goes offline, a node must replicate the keys it is responsible for. Heuristics such as replicate to the next k nodes can be applied for this purpose.

A node may also need to replicate its keys when its neighbors are updated.

- [s] List of key/value pairs

7 Conclusion

In this paper, we have defined DHT-independent and DHT-dependent features of various DHT algorithms and presented a comparison of Chord, Pastry and Kademlia. We then described DHT operations and their information requirements. We are working on designing a simple protocol to implement these operations.

8 Acknowledgements

This draft reflects discussions with Kundan Singh.

References

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.

[3] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, pp. 41–53, Jan. 2004.

[4] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *IPTPS'01: Revised Papers from the First International Workshop on Peer-to-Peer Systems.* London, UK: Springer-Verlag, 2002, pp. 53–65.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications.* New York, NY, USA: ACM Press, 2001, pp. 161–172.

[6] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing.* New York, NY, USA: ACM Press, 1997, pp. 654–663.

[7] "eDonkey. http://www.edonkey.com/."

[8] D. Willis, D. Bryan, P. Matthews, and E. Shim, "Concepts and terminology for peer-to-peer SIP," Internet Draft, June 2006, work in progress.

[9] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems," *Communications of the ACM*, vol. 46, no. 2, pp. 43–48, 2003.

[10] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04)*, Boston, Massachusetts, June 2004.

[11] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," in *Workshop on Algorithms and Data Structures*, 1989, pp. 437–449.

[12] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a common api for structured peer-to-peer overlays," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03)*, Berkeley, CA, 2003.

[13] K. Singh and H. Schulzrinne, "Data format and interface to an external peer-to-peer network for SIP location service," Internet Draft, May 2006, work in progress.

[14] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "Opendht: a public dht service and its uses," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications.* New York, NY, USA: ACM Press, 2005, pp. 73–84.