SIPPING WG S. Baset
Internet-Draft H. Schulzrinne
Expires: April 19, 2007 Columbia University
E. Shim
Panasonic
October 16, 2006

A Protocol for Implementing Various DHT Algorithms
draft-baset-sipping-p2pcommon-00

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.

Table of Contents

1. Introduction

Over the last few years a number of distributed hash table (DHT) algorithms [7][8][9][10] have been proposed. These DHTs are based on the idea of consistent hashing [10] and they share a fundamental principle: route a message to a node responsible for an identifier (key) in O(log_{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 with values 2, 4, 16 and so on. Identifiers are logically considered to be arranged in a circle in Chord [7], Kademlia [9] and Pastry [10] and a routing metric may determine 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 [7], Pastry [9], and Kademlia [10] to define a protocol by which any of these algorithms can be implemented. We have chosen Chord, Pastry and Kademlia because either they are being actively researched (Chord and Pastry) or they have been used in a well-deployed application (Kademlia in eDonkey [15]). 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 used in our comparison of DHTs and then give a brief description of Chord, Pastry and Kademlia. The authors of these algorithms have proposed a number of heuristics to improve the lookup speed and performance such as proximity neighbor selection (PNS) 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. 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.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

Some of the terminology has been borrowed from the P2P terminology draft [16].

P2PSIP Overlay Peer (or Peer). As defined by Willis et al. [16], a P2PSIP peer is a node participating in a P2PSIP overlay that provides storage and routing services to other nodes in P2P overlay and is capable of performing several different 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). As defined by Willis et al. [16], 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.

P2PSIP Peer-ID (or Peer Key). As defined by Willis et al. [16], a Peer-ID is a information that uniquely identifies a peer within a given P2PSIP overlay. In the DHT approach, this is a numeric value in the hash space. We use the term identifier and key interchangeably.

Routing table. A routing table is used by a node to map a key to a peer responsible for it. It contains a list of overlay peer keys and their IP addresses stored against identifiers that are exponentially away from the peer key. Simplistically, a routing table contains logN number of entries where N is the number of nodes in the system.

Routing table row. A row in a routing table stores the peer-ID and IP address of peer(s) against a routing table key. The routing table key is computed from the peer key according to a particular routing metric e.g., a Chord peer computes its ith routing table key by performing the following modulo arithmetic:

(PeerKey + 2^{i-1}) mod 2^{M}

where M is the key length and i is between 1 and M. A peer can only store in its ith row a reference to a peer whose key lies between (i) and (i+1) rows of the routing table.

Routing table row interval or range. The interval for row 'i' is defined as the keys that lie between (i) and (i+1) rows according to a particular routing metric.

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 key is `closer' to k than its own key. 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 [12] :

Chord. Numeric difference between two keys. Specifically: (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: a XOR b

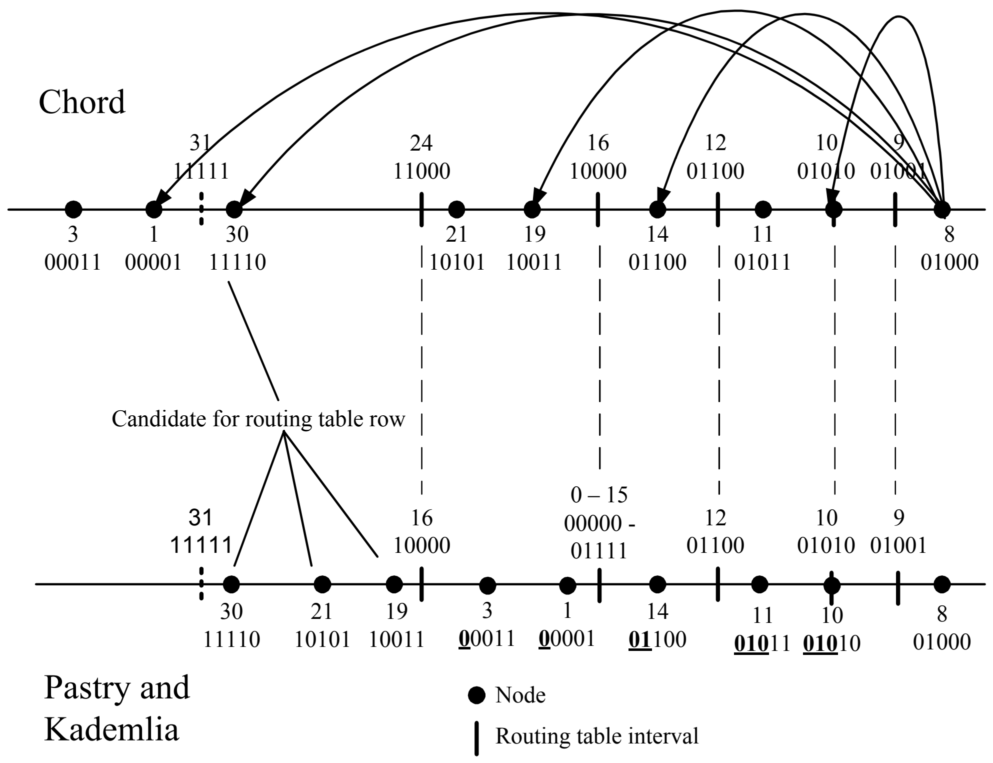
Pastry uses numerical difference when prefix-matching does not match any additional bits and the peers which are closer by prefix-matching metric may not be closer by the numerical difference metric.

3.2. Routing Table Rigidity

There are two ways in which a peer can select a node to fill its ith routing table row. It can be a node whose peer-ID either immediately succeeds or precedes the routing table row interval or it can be any node whose ID lies within the interval. For its ith row, Chord selects a node with an ID which immediately succeeds the interval while Pastry and Kademlia pickup any node with an ID that lies within the 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 1 shows how a peer having the same key 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 ith routing table row, a peer always selects a node with an ID that immediately precedes or succeeds the interval, then the number of such peers is only one. However, choosing any peer whose ID lies within the interval provides more flexibility as the number of candidate nodes increases from one to the number of 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 references to two or more DHT nodes, then it may send a lookup query to both 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. Clearly, there is a tradeoff between sending keep-alives to routing table peers, and sending parallel lookup queries.

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 hop. 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 and forwards it to a peer without replying to the querying peer. This process repeats till the key is found or the query cannot be forwarded which implies that the key does not exist. Rhea [13] 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 provides lower latencies than iterative lookup.

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 [11] 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.

	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 1. Paper specific details of Chord, Pastry and Kademlia.

	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

Table 2. Algorithm specific details of Chord, Pastry and Kademlia.

4.1. Chord

The identifiers or keys in Chord can be logically considered 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 [6]. 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.

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.

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 logically considered 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 [12]. 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 typically 2, 4, 6, 8 etc. The maximum size of routing table is log_{2^b}N x 2^b. 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 log_{2^b}N.

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 each entry i with a common prefix p_{i} is closer to the node (in the network sense) among all other live nodes having a prefix p_{i}. 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 [9]. This method is conceptually similar to Chord's replication of key/value pairs on its successor list.

4.3. Kademlia

Like Chord and Pastry, the identifiers in Kademlia can be logically thought of being arranged on a circle; however the routing is done in a tree-based (prefix-matching) fashion. Each node in Kademlia contains a \emph{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 XOR y. XOR metric is non-Euclidean and it offers the triangle property: d(x,y)+d(y,z) >= 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. The latest incarnation of Chord contains more than one finger entry.

The routing table size is 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 x log_{2^b}N x 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.

	Keep-alive	Lookup	Store	Join	Updating routing table	Updating neighbor nodes
Chord	fix_ fingers()	find_ successor()	N/A	join()	fix_ fingers()	stab()
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

Table 3. DHT specific RPC's

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 can think of 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. [14] 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 et al. [17] defined a XML-RPC based API for DHTs. Their approach is based on OpenDHT [5] 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 by a peer after applying a particular routing 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] 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 [16].

[r] Full or partial routing table of an existing node(s).

[r] List of immediate neighbors.

where [s] is the information sent by the querying peer, [r] is the information received by a peer and [a] is the information appended by a peer to a request before forwarding it to the next hop.

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.

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] Value. A sender may not send the value along with key. It may only send the value only after the peer responsible for the key has been discovered.

[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.

[s] A flag indicating whether the lookup should be performed recursively or iteratively.

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 in a sequential or parallel way. 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] 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.

[r] Value of the key or an indication that key cannot be found.

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.

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. Security Considerations

TBD.

8. References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Petersen, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[3] Bryan, D. and C. Jennings, "A P2P Approach to SIP Registration and Resource Location", draft-bryan-sipping-p2p-01 (work in progress), July 2005.

[5] Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., and H. Yu, "OpenDHT: a public DHT service and its uses", SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications Philadelphia, Pennsylvania, pp. 73-84, 2005.

[6] Pugh, W., "Skip Lists: A Probabilistic Alternative to Balanced Trees", Workshop on Algorithms and Data Structures pp. 437-449, 1989.

[7] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D., Kaashoek, M., Dabek, F., and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", IEEE/ACM Transactions on Networking (To Appear).

[8] Ratmasamy, S., Francis, P., Handley, M., Karp, R., and S. Shenker, "A Scalable Content-Addressable Network", Proc. ACM SIGCOMM (San Diego, CA), pp. 161-172, August 2001.

[9] Rowstron, A. and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), March 2002.

[10] Maymounkov, P. and D. Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric", IPTPS'01: Revised Papers from the First International Workshop on Peer-to-Peer Systems London, UK: Springer-Verlag, pp. 53-65, March 2002.

[11] Karger, D., Lehman, E., Leighton, T., Panigraphy, R., Levine, M., and D. Lewin, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web", STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing , 1997.

[12] Balakrishnan, H., Kaashoek, F., Karger, D., Morris, R., and I. Stoica, "Looking up data in P2P systems", Communications of the ACM vol. 46, no. 2, pp. 43-48, 2003.

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

[14] Dabek, F., Zhao, B., Druschel, P., Kubiatowicz, J., and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays", Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS03) Berkeley, California, February 2003.

[15] "eDonkey", <http://www.eDonkey.com>.

[16] Willis, D., Bryan, D., Matthews, P., and E. Shim, "Concepts and Terminology for Peer-to-Peer SIP", draft-willis-p2psip-concepts-02 (work in progress), October 2006.

[17] Singh, K. and H. Schulzrinne, "Data format and interface to an external peer-to-peer network for SIP location service", draft-singh-p2p-sip-00 (work in progress), May 2006.

Authors' Addresses

Salman A. Baset
Dept. of Computer Science
Columbia University
1214 Amsterdam Avenue
New York, NY 10027
USA

Email: salman@cs.columbia.edu

Henning Schulzrinne
Dept. of Computer Science
Columbia University
1214 Amsterdam Avenue
New York, NY 10027
USA

Email: hgs@cs.columbia.edu

Eunsoo Shim
Panasonic Princeton Laboratory
Two Research Way, 3rd Floor
Princeton, NJ 08540
USA

Email: eunsoo@research.panasonic.com

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.