Storing and locating mutable data in structured peer-to-peer overlay networks

Antony Chazapis and Nectarios Koziris

National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
{chazapis, nkoziris}@cslab.ece.ntua.gr

Abstract. Structured peer-to-peer overlay networks or Distributed Hash Tables (DHTs) are distributed systems optimized for storage and retrieval of read-only data. In this paper we elaborate on a method that allows them to manage mutable data. We argue that by altering the retrieval algorithm of DHTs, we can enable them to cope with data updates, without sacrificing their fundamental properties of scalability and fault-tolerance. We describe in detail and analyze an implementation of a Kademlia network capable of handling mutable data. Nevertheless, the corresponding protocol additions can easily be applied to any DHT design. Experimental results show that although the process of managing and propagating data updates throughout the network adds up to the total cost of the lookup operation, the extra network utilization can be exploited in favor of overlay resilience to random node joins and failures.

1 Introduction

Structured peer-to-peer overlay networks represent a large class of distributed systems that focus on providing scalable and fault-tolerant key-value pair storage in a continuously changing pool of unreliable and unrelated networked computers. These systems, which are most commonly referred to as Distributed Hash Tables (DHTs), store read-only copies of key-value pairs at various nodes of the network in a way that enables them to implement an optimized "lookup" function: Given a specific key, they will return the corresponding value in a very small number of steps - usually proportional to the logarithm of the number of total participants in the network.

In fact, all DHTs use a common methodology to solve the problem of distributing the network workload and data to participating nodes: Assuming a large virtual identifier space of a predefined structure, both nodes and data are given unique IDs that correspond to specific locations within. The algorithm producing the IDs must guarantee that nodes and data will be uniformly distributed in the identifier space. Then, each node takes on the responsibility of storing values and managing lookup queries that refer to data with IDs "close" to its own. The notion of closeness depends on the details of each specific DHT implementation, as does the arrangement of the identifier space. For example, Kademlia

[1] uses an XOR metric to measure distances between the leaves of a binary tree, while Chord [2] places all IDs around a clockwise circle. A brief description of the structures and algorithms used by various structured peer-to-peer systems is provided by Balakrishnan et. al [3].

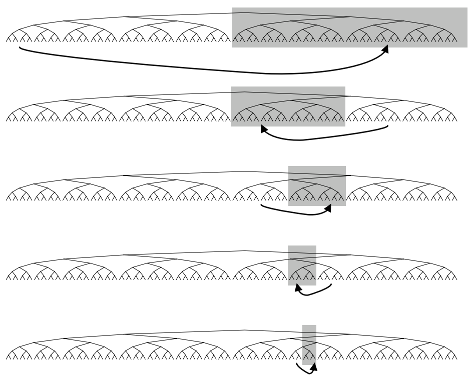


Fig. 1. Looking up a key in a Kademlia overlay

DHTs were designed from ground up to be read-only distributed key-value list storage systems. Nevertheless, in this paper we explore an algorithm based on versioning that enables the use of Kademlia as a mutable data storage and retrieval facility. In the following section we analyze the changes made to the Kademlia protocol and discuss on the differences between our system and analogous designs that have been proposed by the research community. Experimental results from an early implementation suggest that our protocol additions enable data to be inherently changed in the DHT with very few implications on the performance and scalability properties of the overlay. Before concluding, we present thoughts on applications and future extensions of our design.

2 Design

A solution to the problem of storing mutable data in a DHT is presented by the designers of Ivy [4]. Ivy is a distributed file system, that uses the read-only key-value pair storage infrastructure of the peer-to-peer network to create an archive of linked change records. Each value contains actual data, in the form of changes done to the file system, and the identifier of the next read-only tuple in the linked list (previous set of changes). By knowing the key of the latest list item put in the network, an external system can walk the list (called “log”) from the most recent to the oldest change. An analogous design is followed by OceanStore [5], which implements a file management layer on top of an underlying Tapestry [6] network. In such systems, there may be a need to go through hundreds of key-value lookups in the DHT in order to find the latest aggregate value, which would incur an intolerable cost in terms of network messages. Even more, log records never get deleted as they are needed for recovery in case of network failures and there is no straight-forward way to identify which key-value pairs are no longer needed in the list. Using the model of Ivy or Oceanstore, one can store mutable data in a DHT, assuming there is another distributed system coordinating the updates of the latest keys inserted in the peer-to-peer overlay.

In the contrary, we would like to enable inherent mutable data storage at the level of each individual key-value pair stored in the system. Related work in the field has investigated the applicability of distributed mutual exclusion protocols [7] in DHTs. In order to update a key-value pair, a node must acquire the permission to do so, by requesting the lock from all servers holding replicated instances of the tuple. If the majority replies, the lock is granted and the new value can be propagated to the corresponding servers. Nevertheless, distributed mutex algorithms in general, require that there is a well defined algorithm for identifying which nodes hold replicated values. In DHTs, key-value pairs are replicated to nodes that are close to the ID of the key and cached around the network. It is extremely difficult to trace which nodes hold a particular tuple and provide guarantees that lock violations and deadlocks will not occur in an environment of continuously changing node behavior (which may in turn alter the placement of replicated data tuples). Moreover, if such an algorithm existed, there would be an easy way to compromise the peer-to-peer network's security properties [8].

Instead, we propose a method of handling mutable data in DHTs with a relaxed consistency between replicated instances of key-value pairs. Our design leaves the DHTs *store* operation unchanged, but alters the way the system handles *lookups*. Updates are routed to the overlay by storing the new value to the closest nodes of the tuple. The algorithm uses a version identifier along every data item and tries to "inform" at least the nodes returned by subsequent lookup operations of the change. In DHT semantics, a participating node receiving a *store* command for a key-value pair already present at its local repository, should check if the version of the given data item suggests a new, updated value or not. Latest versions of key-value tuples should always replace local pairs with the same ID. By utilizing versions, values can eventually be updated (or even deleted) without the need to employ new control commands at the peer level.

Nevertheless, the closest nodes to a key-value pair may change over time in an unpredictable manner. The storage procedure by itself does not guarantee that the whole network will be aware of the value change. As we have updated the replicas stored at the nodes returned by the corresponding *lookup* procedure, there is a very high probability that upon subsequent queries for the same key, at least one of the updated peers will be contacted. Thus, data changes can be propagated to nodes when key-value pairs are looked up. This requires that the algorithm for locating data items will not stop when the first value is returned, but will continue until all available versions of the pair are present at the initiator. The querying node will then decide which version to keep and send corresponding *store* messages back to the peers that seem to hold previous or invalid values.

2.1 Implementation

With the above design in mind, we have tweaked the Kademlia protocol to support mutable data storage. While these changes could have been applied to any DHT (like Chord or others), we picked Kademlia as it has a simpler routing table structure and uses a consistent algorithm throughout the lookup procedure. Kademlia nodes also support issuing multiple concurrent requests in each query step (up to α), which reduces the total time needed for queries to complete and helps in identifying and bypassing faulty peers that could stall the process. Also κ, a system-wide parameter, specifies the number of replicas maintained for each data item and controls the size of routing tables.

According to the Kademlia protocol, three RPCs take place in any data storage or retrieval operation: `FIND_NODE`, `FIND_VALUE` and `STORE`. To store a key-value pair, a node will first need to find the closest nodes to the key. Starting with a list of closest nodes from its own routing table, it will send parallel asynchronous `FIND_NODE` commands to the top α nodes of the list. Nodes receiving a `FIND_NODE` RPC should reply with a list of at most κ closest peers to the given ID. The requesting node will collect the results, merge them in the list, sort by distance from the key, and repeat the process of querying other nodes in the list, until all κ closest nodes have replied. Actually, the initiator does not wait for all α concurrent requests to complete before continuing. A new command can be generated every time one of the α inflight RPCs returns new closest nodes candidates. When the list is finalized, the key-value pair is sent for replication to the corresponding peers via `STORE` RPCs. Kademlia instructs that all key-value pairs are republished in this way every hour, and expire in 24 hours from their initial publication.

To retrieve a value from the system, a node will initiate a similar query loop, using `FIND_VALUE` RPCs instead of `FIND_NODE`s. `FIND_VALUE` requests return either a value from the remote node's local repository, or - if no such value is present - a list of at most κ nodes close to the key. In the later case, this information helps the querying node dig deeper into the network, progressing closer towards a node responsible for storing the value at the next step. The procedure stops immediately when a value is returned, or when the κ closest peers have replied and no value is found. On a successful hit, the querying node will also cache the data item to the closest peer in the lookup list that did not return the value, with a `STORE` RPC. Moreover, whenever a node receives a command from another network participant, it will check its local key-value pairs and propagate to the remote peer the ones that are closer to its ID. This guarantees that values are replicated to all of their closest nodes and helps peers receive their corresponding data items when they join the network.

Our modified lookup algorithm works similar to the `FIND_NODE` loop, originally used for storing values in the network. We first find all closest nodes to the requested key-value pair, through `FIND_NODE` RPCs, and then send them `FIND_VALUE` messages. The querying node will check all values returned, find the most recent (all key-value pairs have associated timestamp metadata) and notify the nodes having stale copies of the change. Of course, if a peer replies to the

FIND_VALUE RPC with a list of nodes it is marked as not up to date. When the top κ nodes have returned a result (either a value or a list of nodes), we send the appropriate STORE RPCs. Nodes receiving a STORE command should replace their local copy of the key-value pair with its updated version. Storing a new key in the system is done exactly in the same way, with the only difference that the latest version of the data item is provided by the user, so there is no need to send FIND_VALUE RPCs to the closest nodes of a key (version checking is done by the remote peers). Moreover, deleting a value equals to updating it to zero length. Deleted data will eventually be removed from the system when it expires.

2.2 Discussion

In the original Kademlia protocol, a *lookup* operation will normally require at most log(N) hops through a network of N peers. The process of searching for the key's closest nodes is complementary to the quest for its value. If an "early" FIND_VALUE RPC returns a result, there is no need to continue with the indirect FIND_NODE loop. On the other hand, the changes we propose merge the *lookup* and *store* operations into a common two-step procedure: Find the closest nodes of the given key and propagate the updated value. Cached items are ignored and lookups will continue until finding all nodes responsible for the storage of the requested data item. The disadvantage here is that it is always necessary to follow at least log(N) hops through the overlay to discover an identifier's closest peers.

Thus, getting data is not faster - it costs exactly the same as when storing it. There certainly can not be a way to support such a major change in the peer-to-peer system without paying some cost, either in terms of bytes exchanged or in terms of increased latency required for a result (two benchmarking metrics proposed as a common denominator in evaluating peer-to-peer systems [9]). Nevertheless, the penalty in messages needed by our system to to support mutable data does not come without a gain: In every such operation the latest version of a key-value pair will actually be republished on the network. There is no longer a need to explicitly redistribute data items every hour. Values are automatically republished on every usage, but also explicitly reseeded to the network when an hour passes since they were last part of a *store* or *lookup* operation. Also, by using more messages for retrieving data items, routing tables are updated more frequently and the network becomes even more resilient to failures.

We believe that the changes we propose for Kademlia can easily be adopted by other DHTs as well. There is a small number of changes required and most (if not all of them) should happen in the storage and retrieval functions of the protocol. There was no need to change the way Kademlia handles the node join procedure or routing table refreshes. Of course, every structured peer-to-peer overlay willing to use mutable data should find a new way to correlate keys and values for data items. Most networks up to date use a simple approach where keys are generated from the SHA1 hash of their value. With SHA1 hashes, if the value is to change, so will the key. Keys in networks supporting the *update* command should have a deterministic meaning of their own - much like the one filenames have for file systems - and not be dependent of their corresponding values. Also, there is still a requirement that key-value pairs expire 24 hours after their last modification timestamp. Among other advantages refreshing provides, it is the only way of completely clearing up the ID space of deleted values.

3 Evaluation

In order to study the behavior of the proposed system, we implemented the full Kademlia protocol plus our additions in a very lightweight C program. In the core of the implementation lies a custom, asynchronous message handler that forwards incoming UDP packets to a state machine, while outgoing messages are sent directly to the network. Except from the connectionless stream socket, used for communicating with other peers, the message handler also manages local TCP connections that are used by client programs. The program runs as a standard UNIX-like daemon. Client applications willing to retrieve data from the network or store key-value pairs in the overlay, first connect to the daemon through a TCP socket and then issue the appropriate *get* or *set* operations. All items are stored in the local file system and the total requirements on memory and processing capacity are minimal.

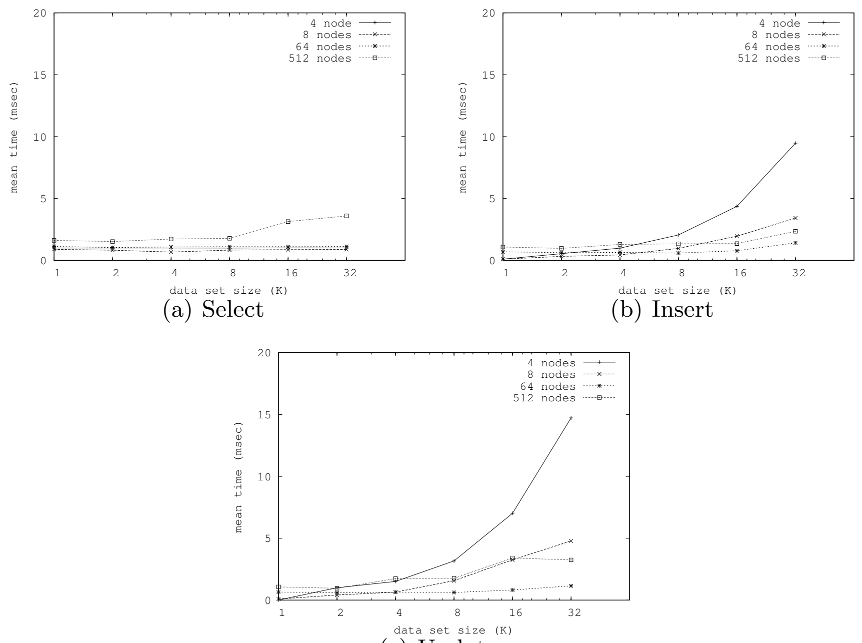
For our tests we used a cluster of eight SMP nodes, each running multiple peer instances. Another application would generate insert, update and select commands and propagate them to nodes in the peer-to-peer network.

3.1 Performance in a static network

To get some insight on the scalability properties of the underlying DHT, we first measured the mean time needed for the system to complete each type of operation for different amounts of key-value pairs and DHT peers. Kademlia's parameters were set to α=3 and κ=4, as the network size was limited to a few hundred nodes.

Figures 2(a), 2(b) and 2(c) show that the prototype needs less than 2 milliseconds to complete a select operation and an average of 2.5 milliseconds to complete an insert operation in a network of 512 nodes with up to 8K key-value pairs stored in the system. The overall system seems to remain scalable, although there is an evident problem with disk latency if a specific node stores more than 8K key-value pairs as individual files in the file system. This is the reason behind the performance degradation of the four node scenario as the amount of mappings increases. As κ has been set to 4, all data items are present at all 4 nodes. When the network has 8K key-value pairs, each node has a copy of all 8K mappings. In future versions of the implementation, we intend on evaluating embedded, lightweight database engines like SQLite for the local storage requirements of each node.

Nevertheless, systems larger than 4 nodes behave very well, since the mean time to complete queries does not experience large deviations as the number data items doubles in size. Also, the graphs representing inserts and updates are



(a) Select (b) Insert

(c) Update

Fig. 2. Mean time to complete operations in a static network

almost identical. The reason is that both operations are handled in the same way by the protocol. The only functional difference is that inserts are done in an empty overlay, while updates are done after the inserts, so the version checking code has data to evaluate.

3.2 Performance in a dynamic network

Our second goal was to measure the performance of the overlay under high levels of churn (random participant joins and failures), even in a scaled-down scenario. Using the implementation prototype, we constructed a network of 256 peers, storing a total of 2048 key-value pairs, for each of the following experiments. Node and data identifiers were 32 bits long and Kademlia's concurrency and replication parameters were set to α=3 and κ=4 respectively. A small value of κ assures that whatever the distribution of node identifiers, routing tables will always hold a subset of the total population of nodes. Also it guarantees that values will not be over-replicated in this relatively small network.

Each experiment involved node arrivals and departures, as long as item lookups and updates, during a one hour timeframe. Corresponding *startup*, *shutdown*, *get* and *set* commands were generated randomly according to a Poisson distribution, and then issued in parallel to the nodes. We started by setting the item update and lookup rates to 1024 $\frac{operations}{hour}$, while doubling the node arrival and departure rates. Initially 64 new nodes were generated per hour and 64 $\frac{nodes}{hour}$ failed. The arrival and departure rates were kept equal so that the network would neither grow nor shrink. Figure 3 shows the average query completion time during a one minute rolling timeframe for four different node join and fail rates. In the simulation environment there is practically no communication latency between peers. Nevertheless, timeouts were set to 4 seconds.

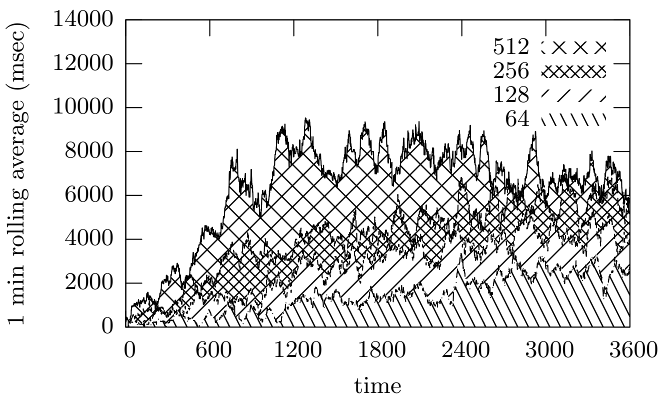


Fig. 3. Time to complete queries while increasing node arrival and departure rates

Handling timeouts As expected, increasing the number of node failures, caused the total time needed for the completion of each query to scale up. High levels of churn, result in stale routing table entries, so nodes send messages to nonexistent peers and are forced to wait for timeouts before they can continue. Kademlia nodes try to circumvent stale peers in *get* operations, as they take α parallel paths to reach the key in question. It is most likely that at least one of these paths will reach a cached pair, while other paths may be blocked, waiting for replies to timeout. Our protocol additions require that caching is disabled, especially for networks where key-value pairs are frequently updated.

Instead, we try to lower query completion times by making nodes dynamically adapt their query paths as other peers reply. In the first phase of the *get* operation, where `FIND_NODE` requests are issued, nodes are instructed to constantly wait for a maximum of α peers to reply from the closest κ. If a reply changes the κ closest node candidates, the requesting node may in turn send more than one commands, thus having more than α requests inflight, in contrast to α in total as proposed by Kademlia. This optimization yields slightly better results in total query completion times, in expense to a small increase in the number of messages. Figure 4 shows a comparison of the two algorithms in a network handling 256 node arrivals and departures per hour.

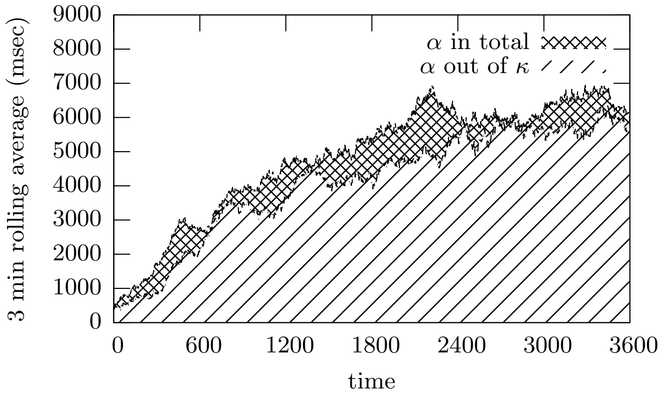


Fig. 4. Dynamically adapting α to changes in the lookup list

Handling lookup failures High levels of churn also lead to increasing lookup failures. Experiment results shown in Table 1 suggest that as the rate of node arrivals and departures doubles, the lookup failure rate grows almost exponentially. In order to prove that the extra messaging cost by our protocol additions can be exploited in favor of overall network fault-tolerance, we reran the worst case scenario (512 node joins and 512 node failures per hour) several times, while doubling the lookup rate from 1024 up to 16384 $\frac{operations}{hour}$. It is evident from the results presented in Table 2, that even in a network with very unreliable peers, a high lookup rate can cause the corresponding failure rate to drop to values less than 1%. This owes to the fact that lookup operations are responsible for propagating key-value pairs to a continuously changing set of closest nodes, while helping peers find and remove stale entries from their routing tables.

Table 1. Increasing node node arrival and departure rates

$\frac{nodes}{hour}$	64	128	256	512
Failures	0	2	32	154
Rate	0.00%	0.19%	3.12%	15.03%

The initial high failure rate is also dependent on the way Kademlia manages routing tables. When a node learns of a new peer, it may send corresponding values for storage, but it is not necessary that it will update its routing table. For small values of κ and networks of this size, routing tables may already be full of other active nodes. As a result, lookups may fail to find the new closest peers to a key. A dominant percentage of lookup failures in our experiments were caused by nodes not being able to identify the latest closest peers of a value. Also, Kademlia's routing tables are designed to favor nodes that stay longer in

Table 2. Increasing the lookup rate

$\frac{operations}{hour}$	1024	2048	4096	8192	16384
Failures	162	106	172	126	84
	131	91	137	80	58
	163	63	145	116	106
	143	61	130	120	87
Rate	15.82%	5.17%	4.19%	1.53%	0.51%
	12.79%	4.44%	3.34%	0.97%	0.35%
	15.91%	3.07%	3.54%	1.41%	0.64%
	13.96%	2.97%	3.17%	1.46%	0.53%

the network, but the random departure scheme currently used by our simulation environment does not exploit this feature.

4 Future work

A DHT capable of handling mutable data can provide the basis of large-scale distributed mass storage and retrieval systems. Key-value pairs can either hold data or pointers to data. For example, peer-to-peer file sharing systems over mutable DHTs can follow a layered approach, deploying a distributed transfer protocol (like the one used by BitTorrent [10]) over an overlay storing a list of pointers to servers holding the actual files. BitTorrent requires that a centralized server (called "tracker") for each file is responsible for holding the list of peers that have started or completed downloading it. This list can be stored as a value in our peer-to-peer system. Dynamic changes in the list may produce inconsistencies, but the upper layer protocol will automatically bypass faulty entries - what really matters is that the list should contain at least one pointer to a valid server holding the file. Other similar applications could as well benefit from this approach. For example, the Grid Replica Location Service, manages a distributed catalog of filename to replica location mappings [11] without imposing any consistency requirements on the update mechanism of the catalog. The Replica Location Service may return a collection of replica locations to applications containing a number of false positive entries. The only requirement is that at least one entry should be correct. We plan on making our implementation work as a core component in this class of programs.

5 Conclusion

A thorough transformation of the default DHT *lookup* operation, has allowed us to implement and evaluate a structured peer-to-peer overlay network capable of handling mutable data. The new lookup procedure is coupled with a simple version management algorithm that handles the reseeding of the latest data updates to the network. Experimental results on a Kademlia-based implementation, support the effectiveness of the protocol changes and additions required, even in networks with a very high rate of changes in node membership status. Moreover, the cost of administering data updates is kept to a minimum, while allowing the overlay network to maintain its inherent characteristics and advantages. Scalability is preserved, as the propagation of data updates happens in a decentralized fashion and the extra messages needed for the maintaining data changes are used by peers to refresh their routing tables and preserve the system's overall tolerance to failures. We believe that a DHT allowing data updates, could be employed by many current and future large-scale distributed applications.

References

1. Maymounkov, P., Mazières, 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 (IPTPS'02), Cambridge, MA (2002)
2. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM Press (2001) 149–160
3. Balakrishnan, H., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Looking up data in p2p systems. Communications of the ACM **46** (2003) 43–48
4. Muthitacharoen, A., Morris, R., Gil, T.M., Chen, B.: Ivy: A read/write peer-to-peer file system. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Boston, MA (2002)
5. Kubiatowicz, J., Bindel, D., Chen, Y., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.: Oceanstore: An architecture for global-scale persistent storage. In: Proceedings of ACM ASPLOS, ACM (2000)
6. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **22** (2004) 41–53
7. Lin, S.D., Lian, Q., Cheng, M., Zhang, Z.: A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems. In: Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04), San Diego, CA (2004)
8. Hazel, S., Wiley, B.: Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems. In: Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02), Cambridge, MA (2002)
9. Li, J., Stribling, J., Gil, T.M., Morris, R., Kaashoek, M.F.: Comparing the performance of distributed hash tables under churn. In: Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04), San Diego, CA (2004)
10. Cohen, B.: Incentives build robustness in bittorrent. (Available online: http://bittorrent.com/)
11. Chervenak, A., Deelman, E., Foster, I., Guy, L., Hoschek, W., Iamnitchi, A., Kesselman, C., Kunszt, P., Ripeanu, M., Schwartzkopf, B., Stockinger, H., Stockinger, K., Tierney, B.: Giggle: a framework for constructing scalable replica location services. In: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press (2002) 1–17