A peer-to-peer replica magement service for high-throughput Grids

Antony Chazapis, Antonis Zissimos and Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15773, Greece
e-mail: {chazapis, azisi, nkoziris}@cslab.ece.ntua.gr

Abstract— **Future high-throughput Grids may integrate millions or even billions of processing and data storage nodes. Services provided by the underlying Grid infrastructure may have to be able to scale to capacities not even imaginable today. In this paper we concentrate on one of the core components of the Data Grid architecture - the Replica Location Service - and evaluate a redesign of the system based on a structured peer-to-peer network overlay. We argue that the architecture of the currently most widespread solution for file replica location on the Grid, is biased towards high-performance deployments and can not scale to the future needs of a global Grid. Structured peer-to-peer systems can provide the same functionality, while being much more manageable, scalable and fault-tolerant. However, they are only capable of storing read-only data. To this end, we propose a revised protocol for distributed hash tables that allows data to be changed in a distributed and scalable fashion. Results from a prototype implementation of the system suggest that Grids can truly benefit from the scalability and fault-tolerance properties of such peer-to-peer algorithms.**

Index Terms— **Grid, Data Grid, data management, replica location service, peer-to-peer networks, distributed hash tables**

I. Introduction

The Grid is a wide-area, large-scale distributed computing system, in which remotely located, disjoint and diverse processing and data storage facilities are integrated under a common service-oriented software architecture [1], [2]. In the hardware layer, a Grid may be comprised of any component that can connect to a shared network and provide the necessary software-level services to be remotely used and administered. Individual computers, clusters, computing farms, network-attached storage arrays, tape libraries or even specialized sensors and scientific instruments can all be part of a single Grid. The software infrastructure of the Grid - the Grid middleware - is responsible of providing the mechanisms of fair and secure resource sharing among the end users of the system. Furthermore, Grid users are organized in "Virtual Organizations". The Virtual Organization is a fundamental Grid structure, with the purpose of enabling the collaboration between multiple mutually distrustful participants. The participants' degree of relationship may be varying to none and their collaborations are based on resource sharing in order to achieve a common goal.

One of the Grid's most essential and critical components is the data management layer. Pioneering Grid efforts [3] were early faced with the problem of managing extremely large-scale datasets - in the order of petabytes - shared among broad and heterogeneous end user communities. It was essential to design a system architecture capable of meeting these advanced requirements in the context of the Grid paradigm. The proposed Data Grid architecture [4] allows the distributed storage and accessibility to a large set of shared data resources, by defining a set of basic data services interacting with one another in order to expose well known, file-like APIs and semantics to end user applications and other higher-level Grid services. In the Data Grid framework, "data" can be anything: From small text files, to big video streams or huge scientific experiment inputs/outputs.

One of the core building blocks of the Data Grid architecture is the Replica Location Service. The Grid environment may require that data is to be scattered globally due to individual site storage limits, but also remain equally accessible from all participating computing elements. In such cases, it is common to use local caching of data to reduce the network latencies that would normally add up as a constant overhead of remote data access operations. In Grid terminology, local copies of read-only remote files on storage elements are called "replicas" [5], while applications running on the Grid request such local file instances through specialized Grid data management services. To work with a file, a Grid application must first ask the Replica Location Service to locate corresponding instances of the requested item so that if a local replica already exists, the application can use normal file semantics to access its contents. In the case only remote copies are found, another component of the Data Grid can take on the responsibility of copying the remote data to the local node and update the replica location indices with the position of the new instance. Data replicas help in improving the performance of applications that require to frequently access remotely placed information. By replicating data closer to the application, the overall access latency is much shorter and the aggregate network usage is reduced. Moreover, through replica-aware algorithms, data movement services can exploit multiple replicas to boost transfer throughput and data recovery tools can reproduce lost original data from their corresponding replicated instances.

Modern Grid middleware distributions like the Globus Toolkit [6] include replica location and management services, as they have become an integral component of the Grid

infrastructure. Moreover, the techniques adopted by the Grid community over the past years in this direction have evolved significantly. The initial design of a centralized Replica Location Service was swiftly put aside in favor of a distributed approach. The most widespread solution currently deployed on the Grid, namely the Giggle Framework [7], constructs a uniform filename namespace of unique per VO identifiers (logical filenames - LFNs) and manages the mappings of these identifiers to physical locations of files (physical filenames - PFNs). LFNs are used by the applications to locate data, no matter the source of the request or the physical location of the information. PFNs, which are used by the Replica Location Service and other Data Grid services, are structured similar to a URL, describing the access protocol, the site and the path in the site directory structure for a given replica. In order to distribute the replica location data throughout the Grid, Giggle makes use of two main components, the local replica catalogs (LRCs) and the replica location indices (RLIs):

- An LRC maintains information about logical filenames such as access lists, creation date and various other file attributes. It also stores a map of all physical filenames that are replicas of a logical filename (LFN to PFN maps). Given an LFN, the LRC will return the associated PFN set.
- An RLI maintains information about the catalogs and the associated logical filenames. It can find which catalog holds the replica file list for a given LFN (LFN to LRC map).

In a default deployment scenario, each participant of a VO manages an LRC, while the overall orchestration of the RLS is done by a single central RLI per VO. When requirements escalate, multiple RLIs can be deployed in parallel, providing optional coarse-grain load-balancing and fail-over features to the replica location infrastructure. The Giggle Framework instructs that multiple indices and catalogs form a two-level hierarchy, with each LRC linked to multiple RLIs and vice versa. Multiple RLIs can also form tree-like structures.

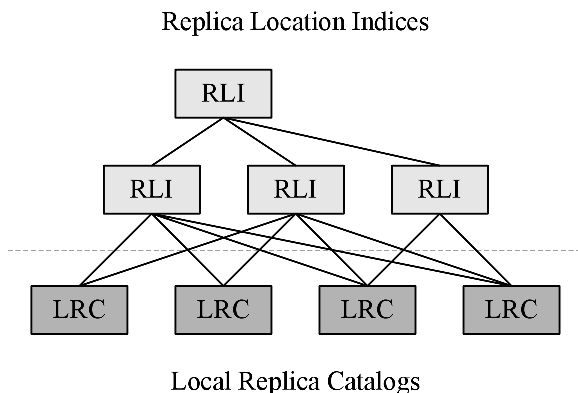

Fig. 1. Giggle deployment example

While scalability had been a major concern during the design of the Giggle architecture, we believe that the distribution approach used may reach its limits, when the number of logical to physical filename mappings or the number of catalogs and indices increase in several orders of magnitude. Giggle is optimized for a high-performance Grid - a Grid that consists of highly-available computing and storage elements, interconnected by high speed networks, running mainly supercomputing applications. The two-level hierarchical structure is not inherently fault-tolerant, as an upper-layer RLI failure, caused by a hardware crash or a network blackout, may bring down the whole system.

Nevertheless, replica management with Giggle may be adequate for current Grid deployments that reside mainly in the high-end scientific area. Grids that interconnect computing and storage resources with scientific instruments, enable scientists from academic or government institutions, although scattered world-wide, to collaborate on a common task, usually the study of experimental results. These early, “academic” Grids, provide scientists with cutting-edge tools and platforms for information acquisition, storage and processing, but more importantly help researchers identify the architectural requirements and design problems that have to be addressed in order to make a global-scale, business-ready Grid infrastructure a reality. The vision of a commonplace Grid that will service billions of end users daily, by providing them ubiquitous access to a vast range of public and private information and services, is realizable, but not as yet feasible. The future, high-throughput Grid may be constituted of millions of independent, interconnected computers, all contributing their unused cycles to the processing needs of a world-wide community. We believe that in order to scale the Grid to these numbers, there is a need to delegate the execution of some of its core services to the edges of its infrastructure. In a network of millions and billions, a task distributed to tens or even hundreds still acts as a “centralized” resource.

We propose that a truly scalable solution to the file replica location problem could be based on the usage of a structured peer-to-peer overlay network. The next section of this paper includes some comments on Giggle’s design limitations, while in the following sections, we concentrate on the observation that although peer-to-peer systems were designed for distributed and scalable lookups, they need a built-in *update* mechanism in order to fit in the requirements set by the Replica Location Service design. We analyze the problems associated with supporting such operations in distributed hash tables, look at previous designs that try to solve the problem by integrating a data management layer on top of a read-only peer-to-peer overlay and propose an algorithm to enable inherent mutable data storage and management in the peer-to-peer network level. In addition, we present how our algorithm can be incorporated into a simple distributed hash table protocol, discuss on the method and evaluate its merits, elaborating on performance results from an early implementation. This paper is concluded with references to related work in the area and thoughts on future work in the same direction.

II. Limitations of the Giggle Framework

Delving into the internals of the Giggle Framework, one can argue that it cannot offer the scalability and performance needed for a global Grid infrastructure. A Replica Location Service must implement partitioning in both the levels of data

storage and data operations in order to be scalable. The Giggle framework may allow tuning of its index and catalog topology via a variety of parameters, but even in a fully distributed environment, the architecture of the service will only allow data storage partitioning. Operations on data, such as changes in the LFN to PFN mappings, will all be concentrated at the node responsible for storing a particular mapping. As a result, specific LRCs may get overloaded when very popular LFNs require frequent updates of their associated PFN lists.

The Giggle prototype implementation [8] requires that data partitioning is configured manually. If the RLS has huge data sets to handle and storage requirements change, the participating nodes must manually adapt to the new situation by specifying new distribution parameters. Moreover, regardless of partitioning, the catalogs and indices cannot automatically handle a new addition or deletion of a participating catalog or index. In general, Giggle parameters cannot be dynamically changed. Although the designers have envisioned a membership management service that will allow the system to deal with unplanned LRC and RLI joins and failures, the current static configuration adds a tremendous cost in the management of the replica location service. Every time a new entity is added in the network or a system parameter needs update the whole service may have to be reconfigured.

Data partitioning has been incorporated in the Giggle framework, both as a method to achieve scalability and as a technique to reduce network and system utilization when LRCs update RLIs. LRCs are required to refresh RLIs, not only in order to inform them on the latest mapping updates, but also to prevent them from deleting old mappings because of timeouts. The update mechanism between the mesh of indices and catalogs has to be as efficient as possible, as it can limit the scalability of the system. The Giggle prototype leaves data partitioning disabled by default. Instead, it is argued that scalability can be achieved by utilizing a soft state protocol. Either full or incremental, updates are asynchronous, so when an add or delete operation occurs, it is not immediately propagated to the appropriate index server. Moreover, soft updates can be very demanding on the size of the data involved. To reduce the overhead of such transactions, they are compressed using Bloom filters - a lossy compression scheme. Eventually, because of asynchronous updates and lossy compression of data, the requesting clients may get false positive answers and appropriate error-handling mechanisms must be developed at the client-side.

According to the experimental analysis of the prototype implementation, compression of the updates induces performance overheads when the filter is initialized and every time a number of hash functions need to be calculated for a filename. In order to reduce the performance loss, the relational database backend is not used when compression is enabled. Instead, there is a need for a customized in-memory data structure and the Giggle code has to support two different methods for the same function. The code becomes more complicated and the logical and organizational advantages of a database backend are lost. On the other hand, although the database backend offers easy modeling and deployment of catalogs and indices, it requires non-trivial fine tuning (e.g. disabling database flush in MySQL or forcing periodic vacuums in PostgreSQL). When database products used are third-party, these modifications may prove even harder to implement.

To deploy and use the Replica Location Service, a large number of parameters have to be tuned, such as the number of RLIs, the function used to partition the LFN or the Replica Site namespace, the degree of redundancy in the index space, the compression method of the soft state updates and the type of the scheduled updates of the catalogs. So, Giggle is difficult to deploy and manage and it can not automatically adopt to unadvertised subsystem joins or failures. We also believe that the pursuit of scalability has led Giggle to employ complex mechanisms to update the data which will in turn limit the efficiency of data retrieval operations on very large networks. There are currently no performance results of a very large RLS system serving millions or billions of mappings, so there is no practical way to plead for this hypothesis, but we feel that there can be an easier way to implement a Replica Location Service for the Grid, with the help of an already scalable, fault-tolerant and self-configurable peer-to-peer network.

III. Peer-to-peer lookup systems and the Grid

The idea of using a peer-to-peer lookup system for locating file replicas in a Grid environment is not new. Ian Foster, Adriana Iamnitchi *et al.* in [9], [10], recognize that the peer-to-peer and Grid research communities have much in common and even more to learn one from another. Services that rely on a peer-to-peer infrastructure can scale without application and environment specific fine-tuning to millions of peer participants, all of which can use the system simultaneously. The network is designed in a scalable way and its potential grows as more participants join, in contrast to traditional client-server models, where overall network performance degrades as more and more clients try to access the centralized resources. The authors of the Giggle system credit the work being done in peer-to-peer location discovery systems as most relevant to theirs. Actually, all peer-to-peer systems try at least to solve the same basic problem as Giggle: Given a unique global identifier, locate in a distributed and scalable way the resource in question [11]. On top of the location service some of the systems will also provide additional services to the participating peers, such as file downloading or media streaming. It is no coincidence that peer-to-peer systems are usually called “lookup systems”.

Peer-to-peer architectures can fall into two basic categories, depending on the structure of the overlay network produced when nodes join and leave the system. Structured systems, or distributed hash tables (DHTs), such as Kademlia [12], Chord [13], Tapestry [14], CAN [15] and others, impose a specific virtual structure which accommodates peers in particular slots as they join the network. On the other hand, unstructured systems like Gnutella leave the peers free to join in any part of the network and the connection graph formed resembles that of a power-law network [16]. Each family of systems has its own advantages and disadvantages over the other: In structured systems the lookup procedure is highly deterministic (will almost always return a result if there is such a value in the

network) and any operation will almost certainly succeed in a predefined number of steps (usually equal to log(N), where N is the number of participating nodes). In unstructured systems, lookups are performed by flooding the network with messages. While there is a high probability that a query will reach a node that can reply for a specific item, it is not definite that the lookup will succeed. If the item is not popular and is stored only at a node far away from the requesting peer, the lookup message will never reach it. Also, flooding in these systems requires far more messages than in distributed hash tables, thus utilizing the network in the extreme. The only advantage of unstructured peer-to-peer systems lies in their ability to handle free-text search queries efficiently and in very few steps, as they inherit a dominant characteristic of power-law networks [17].

In the Grid environment one is not concerned about searching the file servers for a specific file. This operation is provided by metadata servers [18] or can be hidden in application specific semantics. The problem is how to locate the physical file names (replica identifiers) that may be available, when knowing the logical file name of a resource (a unique per VO file identifier). There is also a strong need that this lookup procedure will complete in the minimum possible steps, while maintaining the scalability and availability properties of the lookup system layer. It is obvious that a centralized server storing (LFN, PFN) tuples would handle the lookup operation in a single step, but this solution would neither be scalable nor fault-tolerant. As more servers are added in the lookup layer and data and queries are distributed among them, more messages are needed to traverse the system hierarchy so to reach the desired mappings.

Structured peer-to-peer systems are designed to service storage and retrieval (lookup) of key-value pairs. Keys are always unique for the whole system and serve as identifiers for values. Most of the distributed hash table implementations generate keys directly from the values by computing the SHA1 hash of the data provided for storage. This method produces uniform distributions of keys in a 160-bit identifier space. As a result of the above, in order to utilize distributed hash tables for file replica lookups in a Grid environment, we have to make the following assumptions:

- One overlay peer-to-peer network will be deployed per VO (a single identifier space).
- A key will not be generated by hashing the value of an item. It should correspond to the hash of the logical filename (LFN) of the resource. It will be the unique identifier complementing all data operations.
- A value for a key will actually be a data structure - a list containing the physical locations of replicas (PFNs) for a given identifier.

Also note that in peer-to-peer terminology, the terms *network* and *overlay* refer to the network of virtual interconnections created between the physical *peers* or *nodes* of the system. The latter can practically be applications running on machines connected to the Grid. Other Grid services and end-user applications manipulate data stored in the context of the peer-to-peer overlay, by interacting with nodes through predefined APIs.

IV. Design

The main problem associated with the usage of a distributed hash table to store file replica locations, lies in the disability of the peer-to-peer network to handle mutable data. DHTs may provide *get* and *set* operations, but there is no straightforward way to update data. When a key-value pair is stored into a DHT it is destined to remain in the overlay unchanged until it expires. This shortcoming, emerged as an effect of a DHT design trade-off. The more these systems are made resilient to failures and random node joins and leaves, the more they lose the ability to trace which node is responsible for storing a specific data item. This is inevitable: In a static network there would be no need to duplicate and cache data. Key-value pairs would be placed in specific locations. In DHTs, key-value pairs are copied to nodes that are "close" to the ID of the key and cached around the network. There is no algorithm that can return the exact location(s) of a key-value pair in a given moment (this is also a prerequisite for peer-to-peer network security [19]).

DHTs are made for building dynamic overlays that store non-frequently changing data. While this may seem sufficient for storing the file contents of a read-only file distribution network, it is not enough to serve the needs of the Data Grid's RLS. The *update* operation is absolutely necessary for storing replica locations, as PFN mappings for a given LFN could change frequently and there should be a way for propagating the modifications throughout the network as soon as possible.

One could employ timeout metadata associated with each key-value pair for changing values in the overlay. Data in DHTs expires after a predefined interval since its initial publication, and it is the responsibility of systems external to the peer-to-peer network to update or delete it. But exploiting timeouts to support mutable data is not a solution. The use of small timeout values and the shift of responsibility for change management to an external system, would create scalability problems, destroy any caching advantages and induce severe network utilization for frequent data updates. Moreover, triggering value changes on a timely basis, would not guarantee immediate propagation of updates. Some lookups would seem successful, but the results would include stale values.

A. *Using logs to store and trace data modifications*

A solution to the problem of storing mutable data in a distributed hash table is presented by the designers of Ivy [20]. Ivy is a distributed file system functioning on top of a structured peer-to-peer network. All operations on files and their contents are stored in a distributed hash table, arranged in a linked list of changes - a log. Each participant of the file system knows the identifier of the last data item he put in the system, while each data item contains a list of operations done on the file system and a pointer to the next key-value pair (previous set of changes). By traversing the log from the most recent to the oldest item, the file system can "remember" the latest state of each file and directory for a given participant. As a log exists for each participant of the file system, there

is no need to lock files and directories for concurrent usage between different participants.

To use this algorithm in the file replica location scenario, one could store a list of PFN changes for each LFN. Each list item (key-value pair) would contain a PFN, a status (valid/invalid) and an identifier pointing to the next item (older change). There is also a need to store the mutable head pointer of the list in a well-known place. In Ivy, each participant stores his own head pointer locally and consults the distributed hash table only when walking through the list of immutable change records. In analogy, every member of a VO participating in the management of the VO's file replicas could store a pointer to his change list. Nevertheless, Ivy solves one problem but introduces another. The method used for managing changes is completely inefficient. The use of a distributed log limits scalability and performance. There may be a need to go through hundreds of key-value lookups in the distributed hash table in order to find the current mappings for a given LFN, which would incur an intolerable cost in terms of network messages. Even more, Ivy's log records never get deleted as they are needed for recovery in case of network failures and the cost for managing the status of which entries should be deleted could be enormous.

An analogous design is followed by OceanStore [21]. OceanStore implements a file management layer on top of an underlying Tapestry network. Each file created into OceanStore is associated with a DHT-level key-value pair - the *root block*, which contains information about the file and an index to its corresponding *data blocks* (also maintained in the DHT). To update the contents of a file, one must find its root block, as the root block's maintainer is responsible for serializing write and append requests. Each time a file changes, a new *version* is added to the network. This new version is practically a new set of key-value pairs: a new root block pointing to new data blocks - only those that have been altered from the previous instance of the file. OceanStore also supports file replication. There can be multiple instances of a file in an OceanStore network, but one of them has to be tagged as the *primary replica*. Whenever an instance of a replicated file changes, the updates have to be propagated from the primary replica to all other replicas as well. The update model used is very similar to the one utilized by Ivy, although updates are handled at the file - not the participant - level. Also, the overall design is mainly tailored to support file system semantics. If OceanStore was to be used as a basis for the construction of a scalable RLS, one could associate a "file" to each LFN. Its contents would then be the list of all valid PFN mappings. Furthermore, one more index would be required; the directory of the latest root block IDs for each series of "file" changes. In an hierarchical naming model, this index could be the parent "directory" containing the file, but in flat naming schemes, using another catalog to find replica locations would prove inefficient.

B. Enabling mutable data storage

The ideal solution would be to enable mutable data storage at the level of each individual key-value pair stored at the peer-to-peer system. We argue that this could be done with a very simple addition to the basic distributed hash table algorithm. DHTs may distribute the data in numerous peers of the system, but the only important nodes for every key-value pair are the ones returned by the lookup procedure. If we change the value in these nodes there is a very high probability that upon subsequent queries for the same key, at least one of the updated ones will be contacted. Of course this is not enough, as the network is not a static entity and the nodes responsible for a specific key-value pair storage change over time. DHTs support dynamic node arrivals and departures, so storage relationships between data items and nodes may be altered over time in an unpredictable manner.

As a consequence, every *lookup* should always query all nodes responsible for a specific key-value pair, compare the results based on some predefined version vector (indicating the latest update of the value) and propagate the changes to the nodes it has found responsible for storage but not yet up-to-date with the latest value. This requires that the algorithm for locating data items will not stop when the first value is returned, but 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 older or invalid values. Updates could therefore be implemented through the predefined *set* operation, as version checking would also be done by nodes receiving *store* commands. The latter should check their local storage repositories for an already-present identifier, and if there is a conflict, keep the latest version of the two values in hand. A simple data versioning scheme could be accomplished by using timestamp indicators along every key-value pair.

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 relies on a XOR operation between identifiers to find which nodes are responsible for storing a specific key-value pair. As in any DHT, Kademlia's peers and data items have identifiers from the same address space. XOR is used as the *distance function*, to indicate which are the closest nodes to a given key. By default, when a node of a Kademlia network is instructed to lookup a value through the network, it will issue α parallel queries to the κ closest peers it is aware of, and continue the process as long as no value is returned or it keeps learning of peers even closer to the requested target key. The system-wide parameter κ, also specifies the number of copies maintained for each data item and controls the size of routing tables in peers. Both α and κ variables are set at each participating node and affect only local service performance.

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 copied to the corresponding peers via STORE RPCs. Kademlia instructs that all original 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_NODEs. 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 there are copies of values to all of their closest nodes and helps peers receive their corresponding data items when they join the network.

In the scaled-down example of a Kademlia network shown in Figure 2, both nodes and key-value pairs are mapped to a common 4-bit identifier space. The XOR induced topology is easier to understand if the address space is represented as a binary tree. Nodes and key value pairs are treated as the leaves of the structure, while each node has more routing information for near subtrees and stores items closer to its corresponding leaf. For κ = 2, a data item will be stored at least at its two closer nodes (k3 is stored at n2 and n3). Another node can start locating it in the system by asking a close peer for the item's key. If the remote node can not return a result, it will instead answer with a list of nodes that are even closer to the requested identifier. By repeating the process, the initial peer will finally reach a node responsible for storing a specific key-value (n4 locates k3, stored at n3, by using the list of closest nodes returned by n1).

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 version 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. Moreover, deleting a value equals to updating it to zero length. Deleted data will eventually be removed from the system when it expires.

C. 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 storing 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.

Nevertheless, the lookup procedure is also used to propagate updated values to the network. So the extra cost in messages is equal to the "price" needed by the infrastructure to support mutable data. 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 various peer-to-peer systems [22]). Moreover, the aforementioned drawback in lookup performance could even be accounted as a feature: distributed hash table nodes generally exploit messages exchanged in favor of updating their routing tables. The more messages, the more fault-tolerant the system gets. Also, there is no longer a need to explicitly redistribute data items every hour, as values are automatically republished on every usage. However, the system will still reseed key-value pairs to the network when an hour passes since they were last part of a *store* or *lookup* operation (effectively propagating updates).

Our algorithm can not guarantee that the latest view of a data item will be its latest version, as much as DHTs in general cannot guarantee that the key-value pairs stored in the network will be there when needed. There is always a percentage of success, bound to many parameters that impact the network's reliability and performance. A peer-to-peer network continues to be a dynamic entity, prone to random node joins and leaves, unexpected network failures and diverse usage patterns. It is obvious that although storage of the latest version is propagated to the closest network participants from the querying node's point-of-view and each node individually tries to inform the nodes it knows closer to a key-value pair when it receives RPCs, a major network breakdown may leave stale information in the system, if all nodes responsible for an updated version's storage fail. It is with very high probability

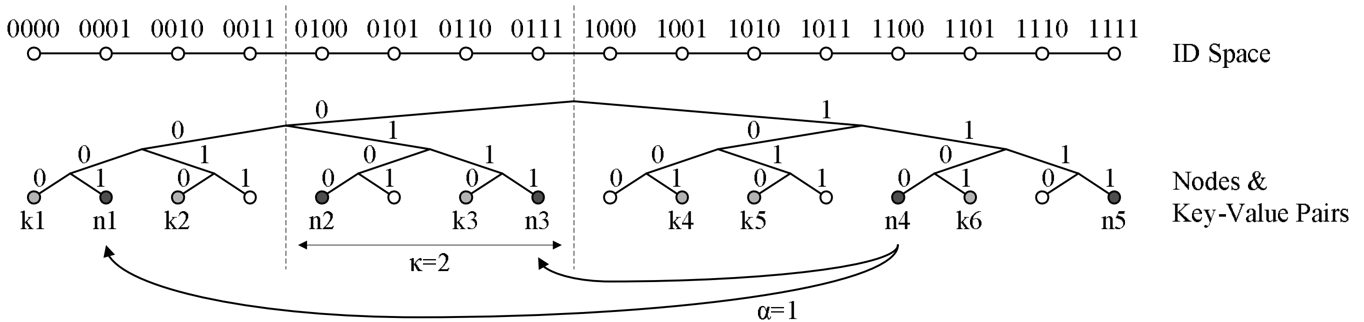

Fig. 2. Scaled-down example of a Kademlia network

though, that at least one node informed on a specific data item update will be found in a subsequent retrieve operation for any key-value pair.

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. Nevertheless, there is still a requirement that key-value pairs expire 24 hours after their last modification. Among other advantages refreshing provides, it is the only way of completely clearing up the ID space of deleted values.

V. Implementation

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 filesystem 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. In the following paragraphs we will present some results from this early system prototype. While our software is not yet complete, it can help us study the basic characteristics and behavior of the peer-to-peer overlay and evaluate our algorithm and the potential it has to support the file replica location needs of Grid applications.

A. 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 3, 4 and 5 show that the implementation takes 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 filesystem. 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.

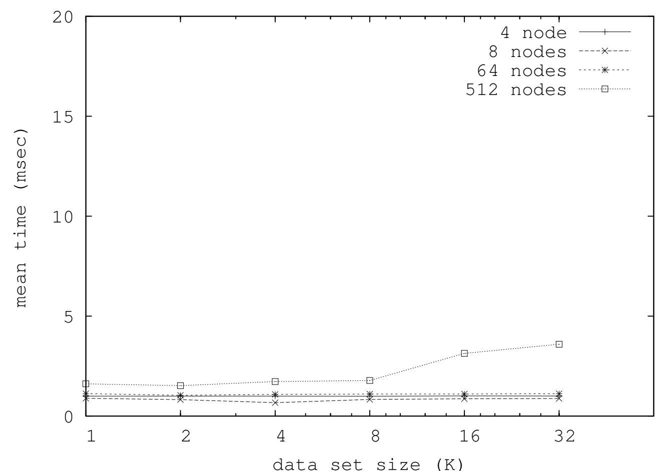

Fig. 3. Select operations

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 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.

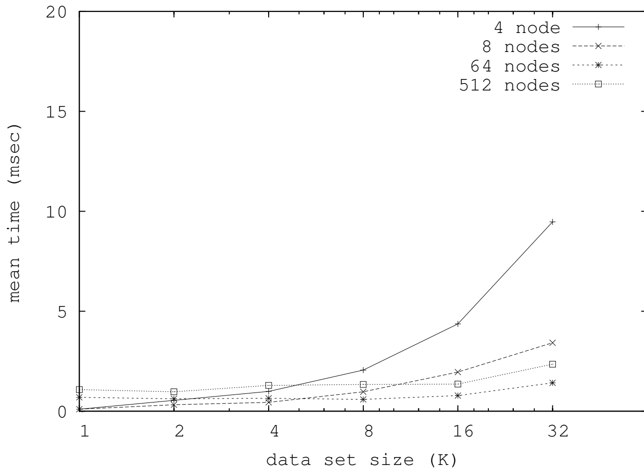

Fig. 4. Insert operations

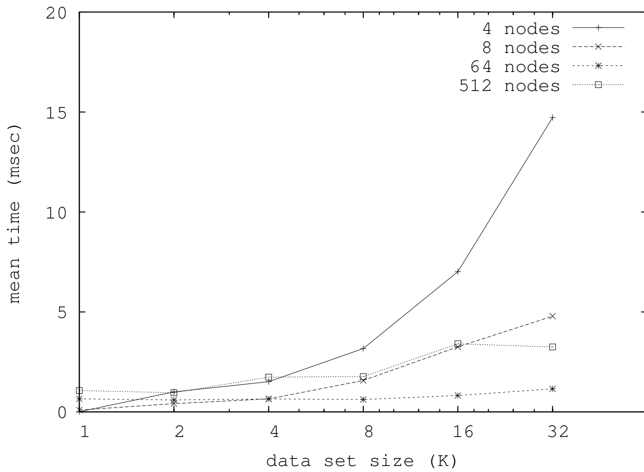

Fig. 5. Update operations

B. 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 6 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.

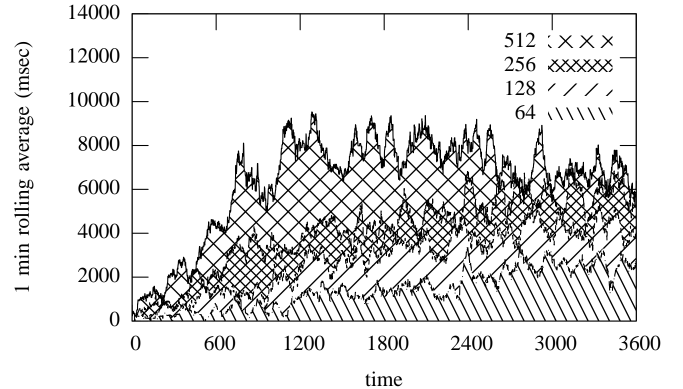

Fig. 6. Time to complete queries while increasing node arrival and departure rates

1) 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.

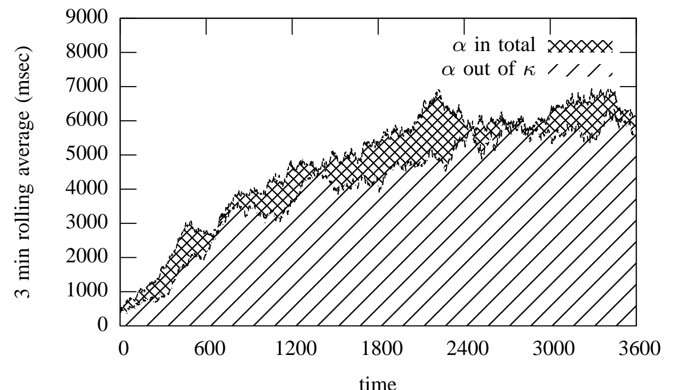

Fig. 7. Adapting the α concurrent requests to changes in the top κ entries of the lookup list

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 7 shows a comparison of the two algorithms in a network handling 256 node arrivals and departures per hour.

2) Handling lookup failures: High levels of churn also lead to increasing lookup failures. Experiment results shown in Table I 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 II, 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 I
LOOKUP FAILURES WHILE INCREASING 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 the network, but the random departure scheme currently used by our simulation environment does not exploit this feature.

TABLE II
LOOKUP FAILURES WHILE 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%

C. Results and future work

The prototype implementation behaves very well in terms of scalability and fault-tolerance, which has allowed us to plan future experiments with much larger network sizes and data set populations. To alleviate the problem with disk storage in future versions of the implementation, we intend on using database-like, single-file storage for local data on each node, or alternatively, an embedded, lightweight database engine like SQLite. Nevertheless, scaling the experiments from a few hundred nodes to orders of magnitude upwards is not straightforward, as it requires special considerations regarding the limits of the underlying simulation hardware and software [23]. We also plan on adding support for Kademlia's *accelerated lookups*.

Moreover, we are working in the direction of coupling our implementation with a flexible peer-to-peer network simulator that will allow us to conduct much larger experiments and measure specific benchmarks relevant to DHT designs [22]. We are focusing on evaluating various aspects of the system, while varying node network and computational performance characteristics. As our prototype approaches production state, we are also considering actual Grid and PlanetLab [24] test deployments.

DHTs normally store successfully retrieved data at the nodes performing the query and take advantage of cached items at subsequent fetch operations. However, our modified Kademlia protocol will not stop the *lookup* operation on a cache hit, so we have disabled all caching in our implementation. A question left open is how to incorporate a caching scheme along our algorithm for distributed mutable data management. If we enable caches there has to be a way of using them without sacrificing the integrity of key-value pairs throughout the network. We are currently investigating various cache management schemes that could fit in as a solution to this problem. There is a need to invalidate caches throughout the network on every data item update. On the other hand, we could just enable caches with small timeouts, especially for replica location environments where *lookups* are much more frequent than *stores* and strict data consistency is not a must.

Another open problem we are looking forward to address in future work is security. The Grid software infrastructure provides advanced security services which we would like to incorporate in our application.

VI. RELATED WORK

Peer-to-peer overlay networks and corresponding protocols have already been incorporated in other RLS designs. In a recent paper [25], Min Cai *et al.*, have replaced the global indices of Giggle with a Chord network, producing a variant of Giggle called P-RLS. A Chord topology can tolerate random node joins and leaves, but does not provide data fault-tolerance by default. The authors choose to replicate data in the *successor set* of each *root node* (the node responsible for storage of a particular mapping), effectively reproducing Kademlia's behavior of replicating data according to the replication parameter κ. In order to update a specific key-value pair, the new value is inserted as usual, by finding the *root node*

and replacing the corresponding value stored there and at all nodes in its *successor set*. While there is a great resemblance to this design and the one we propose, there is no support for updating key-value pairs directly in the peer-to-peer protocol layer. It is an open question how the P-RLS design would cope with highly transient nodes. Frequent joins and departures in the Chord layer would require nodes continuously exchanging key-value pairs in order to keep the network balanced and the replicas of a particular mapping in the correct successors. Our design deals with this problem, as the routing tables inside the nodes are immune to participants that stay in the network for a very short amount of time. Moreover, our protocol additions to support mutable data storage are not dependent on node behavior; the integrity of updated data is established only by relevant data operations.

In another variant of an RLS implementation using a peer-to-peer network [26], all replica location information is organized in an unstructured overlay and all nodes gradually store all mappings in a compressed form. This way each node can locally serve a query without forwarding requests. Nevertheless, the amount of data (compressed or not) that has to be updated throughout the network each time, can grow to such a large extent, that the scalability properties of the peer-to-peer overlay are lost.

In contrast to other peer-to-peer RLS designs, we envision a service that does not require the use of specialized servers for locating replicas. According to our design, a lightweight DHT-enabled RLS peer can even run at every node connected to the Grid.

VII. Conclusion

We believe that in future high-throughput Grid deployments, core services - such as the RLS component of the Data Grid architecture - should be distributed to as many resources as possible. To this end, services must use distribution algorithms with unique scalability and fault-tolerance properties - assets already available by peer-to-peer architectures. In this paper, we argue that a truly scalable and fault-tolerant Replica Location Service can be based on a structured peer-to-peer design (a distributed hash table).

Nevertheless, a read-only key-value pair storage facility is not adequate to store continuously changing replica location mappings. The basic DHT algorithm has to be modified in some way to enable mutable data storage. We have implemented a prototype of a distributed hash table that will allow stored data to be updated through the basic *set* command. Our protocol additions that enable this new operation are very simple and could easily be applied to any analogous peer-to-peer system. We are currently trying to make the initial implementation even more efficient and would like to evaluate its performance in large scale experiments involving close to real-life situations.

The performance of the RLS depends on the effectiveness of its underlying resource lookup algorithm. We do not expect our DHT-based design to outperform the currently deployed system - Giggle, which is based on an hierarchical distribution model. In the contrary, we expect that high-performance Grid deployments will continue to benefit from Giggle's architecture. However, we doubt that Giggle will be able to scale, in order to cover the needs of an extremely large Grid. As the mesh of catalogs and indices grows, the overall service will experience serious bottlenecks in update operations. Also, it requires tuning of various non-trivial parameters and uses complex data structures and algorithms to distribute the lookup data. Our peer-to-peer RLS can run at multiple machines per site or even every machine of the Grid having a public IP address, as the operational requirements are minimal. Furthermore, the architecture of the network will ensure that as more and more nodes join, the replica location infrastructure will scale in storage capacity without significant losses in lookup performance. DHT systems are proved to be extremely scalable and can provide good fault-tolerance characteristics with very simple deployment and management requirements.

References

[1] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[2] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International Journal of Supercomputer Applications*, vol. 15, no. 3, 2001. [Online]. Available: http://www.globus.org/research/papers/anatomy.pdf

[3] "The datagrid project," http://eu-datagrid.web.cern.ch/eu-datagrid/.

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. Tuecke, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of Network and Computer Applications*, vol. 23, no. 3, pp. 187–200, 2000.

[5] H. Stockinger, A. Samar, K. Holtman, B. Allcock, I. Foster, and B. Tierney, "File and object replication in data grids," *Cluster Computing*, vol. 5, no. 3, pp. 305–314, 2002.

[6] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997. [Online]. Available: ftp://ftp.globus.org/pub/globus/papers/globus.pdf

[7] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney, "Giggle: a framework for constructing scalable replica location services," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 2002, pp. 1–17.

[8] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, "Performance and scalability of a replica location service," in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13'04)*, Honolulu, HI, Jun 2004.

[9] I. Foster and A. Iamnitchi, "On death, taxes, and the convergence of peer-to-peer and grid computing," in *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, Feb 2003.

[10] A. Iamnitchi, I. Foster, and D. C. Nurmi, "A peer-to-peer approach to resource location in grid environments," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11'02)*, Edinburgh, UK, Jul 2002.

[11] 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.

[12] P. Maymounkov and D. Mazières, "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, Mar 2002.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "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, pp. 149–160.

[14] 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.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker, "A scalable content-addressable network," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM Press, 2001, pp. 161–172.

[16] M. Ripeanu and I. Foster, "Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems," in *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, Mar 2002.

[17] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman, "Search in power law networks," *Physical Review E64*, vol. 64, pp. 46 135–46 143, 2001.

[18] S. Fitzgerald, "Grid information services for distributed resource sharing," in *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*. IEEE Computer Society, 2001, p. 181.

[19] S. Hazel and B. Wiley, "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, Mar 2002.

[20] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec 2002.

[21] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *Proceedings of ACM ASPLOS*. ACM, Nov 2000.

[22] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, "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, Feb 2004.

[23] E. Buchmann and K. Böhm, "How to Run Experiments with Large Peer-to-Peer Data Structures," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, Apr. 2004.

[24] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A blueprint for introducing disruptive technology into the internet," in *Proceedings of HotNets–I*, Princeton, NJ, Oct 2002.

[25] M. Cai, A. Chervenak, and M. Frank, "A peer-to-peer replica location service based on a distributed hash table," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Pittsburgh, PA, Nov 2004.

[26] M. Ripeanu and I. Foster, "A decentralized, adaptive, replica location service," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11'02)*, Edinburgh, UK, Jul 2002.