Chapter 5

Replica management services on the Grid: Evolving from a centralized design to a fully distributed, scalable and fault-tolerant peer-to-peer infrastructure

A. Chazapis, A. Zissimos and N. Koziris & P. Tsanakas
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory.

Abstract

The Grid infrastructure enables global-scale collaboration of large scientific communities (Virtual Organizations – VOs) usually relying on a common set of data. As information is not necessarily generated and processed centrally, data resources need to be scattered among various data storage facilities. Data objects may also need to be copied from one facility to the other – "replicated" at a site – in order to be evaluated by local VO participants or unique to the site instruments and computing devices. While the main objective when replicating data to remote processes is to reduce aggregate network usage and boost end-application performance, replicas can also be exploited in favor of resiliency to network and site failures and increased data transfer throughput. Nevertheless, managing replicas in highly distributed data storage environments can be very complex. The processes involved in creating, locating, retrieving and updating replicas must guarantee both scalability and fault-tolerance. In this paper, we investigate the requirements of a Grid-centric, replica-based data storage and retrieval infrastructure, present the current trends and implementations and elaborate on how future architectures can capitalize on peer-to-peer technologies in order to achieve even higher levels of scale, without the need of deploying special data management servers.

1 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

WIT Transactions on State of the Art in Science and Engineering, Vol 5, © 2006 WIT Press
www.witpress.com, ISSN 1755-8336 (on-line)
doi:10.2495/1-84564-055-1/05

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.

A critical component of every Grid middleware implementation is the data management layer. Pioneering Grid efforts [3] were at an early stage 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 large 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 are 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", 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 where 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 [5] 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 [6], follows a multi-tier hierarchical structure, distributing data and queries over global (Grid or VO-wide) and local, site-wide RLS instances. Nevertheless, all implementations followed so far in this direction are optimized for "high-performance" operational environments. Current Grid deployments reside mainly in the scientific area and are used to run supercomputing applications on a mesh of highly-available computing and storage devices, interconnected by high speed networks. In a high-performance environment hardware crashes and network blackouts are rare exceptions and may be sustained by redundant equipment or special backup systems.

High-performance Grids enable scientists from academic or government institutions, although scattered world-wide, to collaborate on a common task; usually the study of experimental results. But the Grid concept is not only about how to interconnect computing and storage resources with specialized instruments. Early, "academic" Grids, may provide VOs 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. According to this model, current Grid middleware is a product of evolution. It uses software architectures and algorithms that have evolved in time to compensate for new requirements set from past Grid deployments. However, 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, although realizable, is 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. Mass-scale applications for the Grid have not been designed yet, although there are some primitive examples of what processing resources may be available in a global-scale computing scenario [7, 8].

Without doubt, services provided by the underlying Grid infrastructure may have to be able to scale to capacities not even imaginable today. Recent experiences, like the Grid set up to organize experiments conducted in the Large Hadron Collider facility at CERN [9], designate Grid data management as a major priority for next-generation Grid developments. The huge data size of the experimental results already requires enhanced Grid capabilities in storage and computation. We believe that in order to scale the Grid to these numbers, there is also 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 popular service distributed to tens or even hundreds of nodes may still act as a "centralized" resource, imposing bottlenecks and penalties on performance. Therefore, redesigns of Grid services for

such deployments may benefit from concepts and algorithms used by peer-to-peer overlay networks. Services that rely on a peer-to-peer infrastructure can scale without application and environment specific fine-tuning to billions of simultaneous participants. The service network is designed in a scalable way and its potential grows as more participants join, in contrast to traditional client-server models, where overall performance degrades as more and more clients try to compete for a slice of the server's finite capacity.

In the context of this paper we concentrate on the Data Grid's Replica Location Service. We elaborate on its key position and role in the Data Grid architecture, discuss on the requirements and specifications it must conform to and present the various implementations that have been proposed by the research community up to date. Then we review concepts and algorithms used by peer-to-peer systems and how they can be used for the benefit of a global-scale RLS. An interesting observation is that a special category of these systems, which are tailored for data lookups in a distributed collection of key-value tuples, can effectively address all needs of a truly scale-proof and fault-tolerant RLS infrastructure. However, *structured peer-to-peer networks* or *Distributed Hash Tables* are only capable of storing read-only information. To this end, we also analyze the complications associated with supporting update operations in DHTs, look at previous designs that try to deal with 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, based on performance results from an early implementation. The behavior of the prototype system suggests that Grids can truly benefit from the scalability and fault-tolerance properties of peer-to-peer algorithms, as these systems can provide solid methods of distributing data to numerous network participants, while being much more manageable, scalable and fault-tolerant than "traditional" distributed designs. The continuous growth of the Grid may require a Replica Location Service that can increasingly manage an enormous amount of data objects and their replicas. Evolution demands the use of a distributed system that can meet future scalability requirements – a guarantee provided by a peer-to-peer infrastructure.

2 The Data Grid architecture

By using a Grid setup, numerous sites around the globe can share their storage and computing facilities, through specialized software utilities that abstract resource interfaces into common, implementation-neutral APIs. The applications that can benefit from the Grid architecture range from highly compute-intensive tasks to projects that require the archival and management of extremely large data sets. Grids provide specialized services to control and access resources providing processing capacity, namely *computing elements*, and information archiving, called *storage elements*. Users and Grid-enabled applications can use these services to request any appropriate resource and exploit it to its potential. In addition, all

operations are governed by policies implemented using corresponding Grid security services. The latter encompass all authentication, authorization and auditing processes required for each step of resource consuming.

The structure of the services needed in order to implement a global-scale data management scheme is presented by the authors of the Data Grid architecture [4]. The "Data Grid" term refers to a wide-area, distributed infrastructure of heterogeneous physical components capable of managing immense amounts of data. The Data Grid must provide neutrality to site-specific data storage and retrieval mechanisms and allow policies that control the behavior of components to be defined by users and not service implementors. In practice, this means that data must be accessed in a way that is independent of the actual specific storage system and medium. The Data Grid must allow information to be stored in any combination of file systems, databases and hardware devices. The service architecture must administer the details involved and present a common interface to access, store, transfer and search available data. Moreover, components in the Data Grid architecture must expose all available parameters related to performance to higher-level services and users, as they are responsible for shaping policies according to collaboration-specific agreements.

The Data Grid architecture is structured in two basic horizontal layers. The lowest-level layer includes core components that abstract the basic set of mechanisms related to data and metadata management:

- *Data Services*: This component provides the necessary functions to read, write and get information on *file instances*, which in turn are stored in logical *storage systems*. A file instance is considered as the basic unit of information that can be stored in a storage system. Files must be named in the context of their corresponding storage systems and can be coupled with attributes, such as creation and modification dates, size and access restrictions. On the other hand, storage systems can be implemented as regular file systems or databases that store actual data to hard disks or tapes, or even as specialized distributed high-performance programs that distribute data over a cluster of interconnected physical machines. Data services include utilities to transfer data, either between storage systems and users, or directly from one storage system to another. Corresponding service implementations must also monitor various characteristics of storage systems and expose them to users and higher-level services.
- *Metadata Services*: Analogous services must be provided for metadata management. Metadata is information about the data itself, that can be used to characterize data from several perspectives. Domain independent data attributes stored at metadata repositories for each file can include: Logical storage system independent names, origin and acquisition procedure, data structure and type, access policy, allowed operations (e.g. read, write, append) and so on. In addition, VOs can add domain specific metadata to describe objects, by defining special attributes based on community standards. These attributes may provide better data characterization to aid collaborations in the context of a Virtual Organization. Even individual users

can set their own metadata attributes, in order to support their work. In general, metadata can be used for applying different views of the existing information, assisting in the further analysis and processing of large data collections. A view can be defined as the grouping of data objects on the basis of common attributes and relations.

Other basic services accounted as core components of the Data Grid architecture must address security, resource reservation (QoS), performance measurement and instrumentation issues. On top of the core services layer lies the placeholder for higher-level services. Upper-layer components, which include the Replica Management Service, are designed to deliver integrated solutions to Grid participants. Data replication has been adopted by the Grid community as an efficient mechanism to boost the performance of applications requiring distributed data management and processing capabilities. Appropriate algorithms can exploit the existence of globally distributed identical data copies, in order to improve access latencies and data transfer performance [10, 11]. Replication is, by common consent, the preferred approach to distributed data management on the Grid.

The Replica Management Service is responsible of generating file copies whenever and wherever they are needed by Grid applications. Moreover, it must ensure that replicas are transfered in an optimal way from one site to another and guarantee that if one replica changes, all copies will automatically be updated. The issues that must be addressed by a Replica Management Service are thoroughly discussed by the designers of "Reptor" [12, 13]. Reptor is a replica management system developed to serve the needs of the European DataGrid project (EDG) [14–16]. According to Rector's component-based design, the Replica Location Service is a core building-block of the replica management infrastructure. The role of the RLS is to track down the different physical locations of replicas. It is responsible for the main functionalities of replica management, such as replica creation, deletion and cataloging.

By using the RLS directly, applications can find the locations of all available file copies on the Grid. However, in the context of an ideal Replica Management Service the locations provided by the RLS serve as inputs to advanced Replica Selection Services that optimize file transfers, with respect to network and storage access latencies. Replicas can also be automatically propagated to remote sites based on a subscription model, where interested applications register for files that will be used for processing. The primary goal of all methods is to distribute file copies where they are needed, so that each job can process data locally without the performance penalties imposed by a remote I/O approach. In the case where replicated data need to be updated, there is another component of the RMS that can guarantee consistency across all instances of a file. Replica synchronization can be a very complex problem and although the research community has proposed various mechanisms to address the issues involved [17], most current deployments impose a read-only scheme which greatly simplifies replica management.

3 Timeline of RLS implementations

The Data Grid architecture has emerged as a system design to fulfill the needs of a data-intensive Grid. However, the primary focus of early Grid deployments was in providing scientists and researchers with the necessary tools to handle computational-intensive tasks. It was not until newer instruments were capable of generating huge amounts of detailed data and storage facilities grew in capacity, before it became clear that Grid-based collaborations could evolve around mammoth distributed data sets. The "data-intensive" Grid concept was new at the time, describing a global-scale infrastructure with the purpose of servicing advanced data-centric operations. The new requirements included applying high-performance techniques for organizing and analyzing massive amounts of data in a distributed fashion, while at the same time enforcing authorization, authentication and accounting policies that could enable fair and successful collaborations.

The Replica Location Service is now considered a core component of the Data Grid's Replica Management Services, as we now have a clear understanding of the building blocks needed in order to construct a world-wide data-intensive service platform. Nevertheless, the Data Grid architecture as we know it today is a product of constant evolution. Early data-intensive Grids deployments tried to provide high-performance and high-availability data operations by replicating data, without providing all functions of a complete replica management utility. In the following paragraphs we will see how a specialized RLS emerged as a necessity and discuss on the developments that have shaped the RMS requirements over the years.

3.1 The subscription model

One of the first implementations that addressed the data management needs of a Grid-compatible infrastructure was the Grid Data Management Pilot (GDMP) [18]. The GDMP, developed by the EDG project, was designed to support the Compact Muon Solenoid (CMS) experiment at CERN and is considered as "a pioneer step towards a Data Grid". At the time, the Data Grid was in its early design phase and there were no analogous implementation initiatives that could allow researchers to evaluate performance-related metrics in a production environment evolving around a concrete problem. The CMS experiment generated several terabytes of data per year, organized as objects stored in an object-oriented database (Objectivity). As the data were visualized and analyzed by various parties around the globe, there was an urge to replicate the objects locally at each site. Data replication was not a new concept; it had been used in the context of distributed systems such as databases and file systems. As a matter of fact, GDMP developers could have used inherent database replication systems, but they prompted for a custom solution as it would support better integration with the advanced security and efficient file transfer services provided by the Grid infrastructure. It would also allow them to enforce custom policies for accessing data and managing replicated data instances.

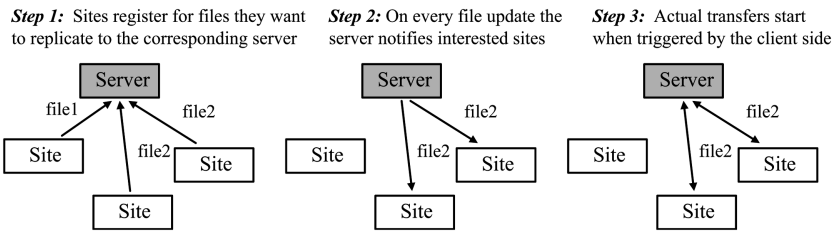

Figure 1: GDMP used a subscription model to maintain data replicas.

The prototype GDMP implementation was based on a subscription model for managing replicas. Sites willing to acquire local copies of Objectivity database files registered to the appropriate server and waited for notifications containing lists of files to be transferred. Actual file transfers were initiated by clients asynchronously. A client should consult the list of new and changed database files and start the required Grid-compatible FTP sessions to get the necessary data. Moreover, because GDMP was realized as a set of command-line tools, it was the responsibility of users to trigger all operations involved in a typical replication scenario. From the server side, a user had to make changes in the database and then use an appropriate command-line tool to automatically notify all subscribers of the event. Then, a user from the client side employed another tool to synchronize his local file replicas with the server (see fig. 1). The lists of changed files on the server and client could also be filtered, allowing each site administrator to implement partial-replication policies. In addition, GDMP handled site failures through simple recovery mechanisms.

GDMP's contribution to the design of the Data Grid architecture was significant. Its modular structure helped in identifying the components required for a next-generation Data Grid. Moreover, the arrangement of objects organized in database files, would later be generalized as files organized in *collections*. By grouping relative data into higher-level containers, there is a need to manage much less metadata for stored information.

3.2 A centralized RLS

Although the initial GDMP prototype did not use any kind of RLS, it soon became evident that such a service would be required in order to implement higher-level utilities capable of exploiting replica location information. Following this line of thought, the first-generation GDMP implementation was extended so that it could interface with the Globus Replica Catalog and GridFTP [19]. Both services were implemented as part of the Globus Toolkit and did not exist at the time of the original GDMP design.

The primary purpose of the Replica Catalog was to maintain mappings between logical names of files and their physical locations. *Logical filenames* (LFNs) were unique identifiers that applications and users could use to refer to

file objects. The Replica Catalog kept track of the replicated instances of LFNs, namely the *physical filenames* (PFNs) of files. PFNs were structured similar to a URL, describing the access protocol, the site and the path in the site directory structure for a given replica. By hiding physical locations of files in a Replica Location Service, applications could use LFNs to work with data, no matter the source of the request or the physical location of the information. LFNs could also be grouped into *collection* objects. Collections represented logical abstractions of data groups and their inclusion in the Replica Catalog came as a result of experience gained from early Grid deployments: Most applications processed groups of multiple related files during their execution. Furthermore, the Globus Replica Catalog could manage metadata for LFNs in the form of attribute-value pairs (*logical file entries*). The catalog was provided with interfaces that could be used directly to manipulate and alter collections, LFNs and their associated attributes. The implementation used an LDAP database backend and replica location information was stored at the level of each individual collection. Given an LFN, the Globus Replica Catalog could find which collections held the file and consult the associated *location objects* of each group in order to get their corresponding *URL constructors*. Each URL constructor identified a unique storage facility and was used to generate the physical locations of files.

The Grid Data Management Pilot exploited the semantics provided by the Replica Catalog in order to manage copies of object-oriented database files. Individual objects could also be transferred from site to site, as long as they were first grouped into temporary database files. In order to avoid complex algorithms required to update replicas, the second-generation GDMP architecture constrained replication to read-only data objects. Moreover, each time an object had to be replicated, the middleware should deal with specific file-format issues. For example, when copying database objects there might be a need to create new schemas in the remote database prior to replication. For the actual data transfer the extended version of GDMP used GridFTP. The last step following every replication was the registration of the new copy in the Replica Catalog.

The overall system architecture was simple enough to support the application requirements of the time. Also, it allowed developers to envision future versions of data movement services that could benefit from the availability of multiple replicas and adapt candidate source files according to individual network bandwidth metrics. Moreover, although the initial GDMP subscription model could still be used, participants were allowed to search the centralized Replica Catalog for files according to their individual interests. This functionality would later be removed from the Replica Catalog and put into a specialized Metadata Catalog. Nevertheless, having all LFNs centrally stored helped in identifying and resolving logical filename conflicts. As we have seen, LFNs had to be unique in the context of a single Grid or VO.

3.3 Distributing the catalog

The Replica Location Service used by GDMP relied on a global, centralized Replica Catalog, which could impose a bottleneck for the whole data management

infrastructure. The next development involved the distribution of the catalog to multiple management parties, according to the method presented by Stockinger & Hanushevsky [20]. As a first step, each local site maintained its own local catalog. Local catalogs were responsible for keeping replica information, namely LFN to PFN mappings, at least for the data objects resident in local storage facilities. Thus, the central catalog was no longer needed to store detailed replica location data, but only the sites responsible for each logical filename. The result was a two-level hierarchical distribution of both data and queries. A global Replica Catalog still existed, but it was only used to redirect requests to an underlying layer of local catalogs. Replication management workload and logical to physical file associations were distributed among all local catalogs and the central server was necessary to be informed only of LFN registrations or deletions.

The global-local distribution schema can be considered to be more Grid-oriented, because of the site autonomy it offers. Processes running in a site can access local replicas without the need of consulting the centralized Replica Catalog. Queries can be answered directly from local resources. Moreover, a simple caching method allows satisfying frequent queries referring to global data objects directly from the local catalog. Except for the obvious advantage of fault-tolerance, a local site can even shape its own policy of which files are accessible to the Grid and which files are not, without the intervention of a central authority. Sites may internally reorganize replica locations for load balancing or backups. In the case of the implementation discussed here, failures were handled by a separate, complementing service that resynchronized all catalogs on a timely basis, to remove stale LFNs from the global server.

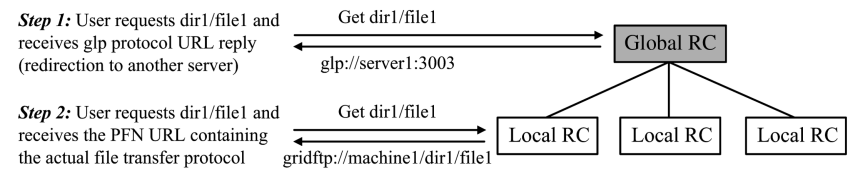

Figure 2: The centralized catalog was distributed to a single global and multiple local servers. Two queries were necessary to get the desired mappings.

The proposed solution was also backwards compatible with older middleware implementations. All catalogs continued to use LDAP database backends, and the data flow of replica location requests remained the same with the addition of an intermediate redirection step. Redirections were realized through the use of a new protocol – the Grid Lookup Protocol (GLP). A query to the central catalog, would result in a physical filename with a glp:// instead of a gridftp:// protocol identifier. The corresponding URL pointed to the location of a local catalog. To get the true physical filenames, the client had to repeat the query to the local site. The GLP protocol was also able to handle Universally Unique Identifiers (UUIDs). In analogy to primary keys in a relational database, each file could be addressed by its unique UUID. Thus, logical filenames could be reused or even changed without the need to update all relevant mappings.

3.4 A new standard: Giggle

The method of distributing the catalog to multiple local instances worked well, but it still relied on a centralized resource – the global catalog – in order to satisfy cross-site replica location requests. The work of Chervenak *et al.* [6, 21] tried to attack this exact problem. They introduced the notion of the Replica Location Service as a distinct component, defined its requirements and described the Giggle (GIGa-scale Global Location Engine) Framework – an RLS architecture that can consist of several global and local services in a multi-tier hierarchy. In analogy to the previous solution, Giggle makes use of two main components in order to distribute the replica location data throughout the Grid: Local, site-wide replica catalogs (LRCs) and global 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).

Using Giggle's components Grid deployments can choose the level of redundancy and distribution of the replica location information. In a default scenario, where each site or VO participant manages an LRC and the overall orchestration of the replica location service is done by a single central RLI, the structure of the RLS network is effectively equal to the aforementioned approach of distributing the Globus Replica Catalog. The advantages of Giggle are evident 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.

Giggle uses a relational database backend instead of an LDAP server, which allows for a greater flexibility in defining the schema of stored information. Moreover, the structure of the hierarchy formed by LRCs and RLIs can be controlled through the definition of a number of deployment parameters. For example, a parameter controls the number of RLIs. There can be a single or multiple RLIs. In fact, the number of RLIs can even exceed the number of local replica catalogs in a highly distributed scheme. In such a case there might also be a need to organize RLIs in a tree hierarchy and partition mappings among global servers, so to avoid the disadvantages of a flat model. Data can be distributed among RLIs according to a parameter defining the desired LFN namespace segmentation function. The policy applied can fall into one of the following categories:

- All LFNs stored by every RLI (no partitioning).
- Random partitioning based on some mathematical function. This will balance the load of the system but does not give any guarantees about the locality of the data.

- Partitioning based on some logical parameter, like the collection that the LFN belongs to. This will guarantee good data locality, but may result into poor load-balancing.

Moreover, Giggle deployments can exploit geographical locality of global indices by partitioning LFN to LRC mappings on the basis of catalog domain names. Another important Giggle parameter controls the degree of redundancy in the index space. There can be no redundancy at all, or even a complete index of replicas across all RLIs. Average values imply mirroring between some of the indices.

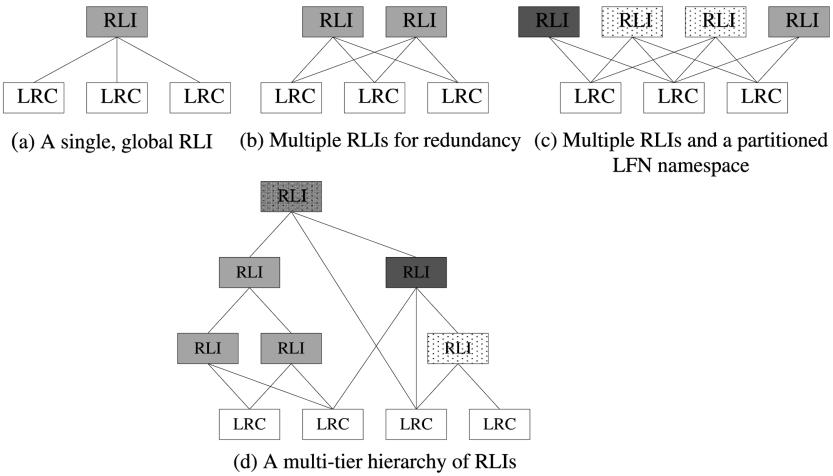

Figure 3: Various possible Giggle deployments. Different RLI colors represent indices that hold different partitions of the LFN namespace.

Giggle uses a complex update scheme to communicate changes between LRCs and RLIs. The catalogs use soft-state update protocols to register new files or renew the time-to-live attributes of files already registered to indices. Full updates can be very demanding in terms of bandwidth, so there is also an option to implement frequent partial updates, communicating only the changes. Moreover, the efficiency of updates can be improved if they are compressed. The proposed compression method is based on Bloom filters. The type and frequency of the updates sent from LRCs to RLIs, and the preferred compression scheme can also be controlled through deployment parameters.

The team behind Giggle has succeeded in grouping all previous work in the field and presenting it as a single, independent solution, optimized for the needs of a scalable and fault-tolerant Data Grid. Giggle is now part of the Globus Toolkit, replacing the obsolete Globus Replica Catalog. Nevertheless, while scalability had been a major concern during the design of the Giggle Framework, it still remains an open question if the distribution approach used will 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 and may prove difficult to deploy and manage in a global-scale infrastructure without highly-available resources. Also, the pursuit of scalability has led Giggle to employ complex mechanisms to update data which may in turn limit the efficiency of replica management operations on very large networks. There are currently no performance results of a very large, production 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.

4 Using a peer-to-peer system as the basis of a scalable RLS

Peer-to-peer networks represent a large class of distributed systems that focus on the construction of a scalable and fault-tolerant *overlay* of interconnected *peers*. In peer-to-peer terminology, the terms *network* and *overlay* refer to the mesh of virtual links created between the physical *peers* or *nodes* of the system. The latter can practically be applications, running on actual machines attached to a common lower-level communication infrastructure, like the Internet. By abstracting the underlying network into a higher-level overlay, peers can "encode" application specific semantics in their corresponding links. In general, there are algorithms that can exploit the overlay design in order to provide optimized resource searching services. Thus, end-user peer-to-peer-based applications can capitalize on the efficiency of the search service so to implement additional utilities, such as file downloading or media streaming.

Recent literature in the field distinguishes peer-to-peer systems into two basic categories, depending on the structure of the overlay network produced when nodes join the system (see fig. 4). Unstructured systems like Gnutella leave the peers free to participate in any part of the overlay and the connection graph

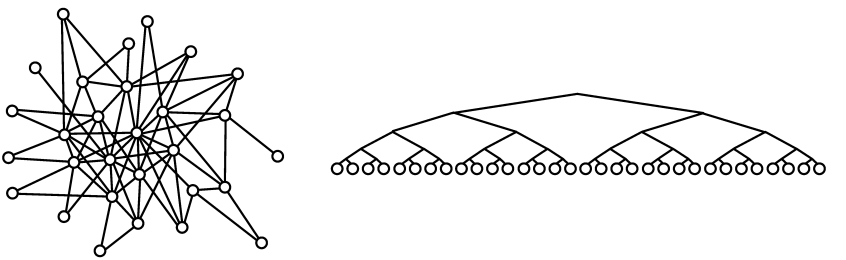

Figure 4: Nodes in a peer-to-peer system can either form *ad-hoc* connectivity graphs or participate in the overlay in a specific part of a virtual structure.

formed resembles that of a power-law network [22] (there is a small number of peers that are highly connected, while the biggest percentage of nodes maintains links to a small subset of network participants). This *ad-hoc* structure can provide peers with the ability to perform free-text search queries efficiently and in very few steps – a dominant characteristic of power-law networks [23]. Nevertheless, searching in these systems requires flooding the network with queries. The flooding technique utilizes the network in the extreme, so unstructured systems are generally inefficient in terms of network bandwidth consumed by their peers. Moreover, 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 search will succeed. If the item is not popular and is stored only at a node far away from the requesting peer, corresponding messages will never reach it.

On the other hand, structured systems such as Kademlia [24], Chord [25], Tapestry [26], CAN [27] and others, impose a specific virtual structure which accommodates peers in particular slots as they join the network. 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 employing a very small number of queries. Links between nodes in a DHT are realized by routing table entries managed by each individual peer, thus nodes decipher data placement information by following specific routes through the overlay structure. 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 (proportional to the logarithm of the number of total participating nodes).

Distributed Hash Tables can guarantee system scalability to very large overlay sizes, as they distribute both the network workload and data to participating peers. The methodology used is common to all DHT implementations: Assuming a large virtual identifier space of a predefined structure, both nodes and data items are given unique IDs that correspond to specific locations within. The algorithm producing the IDs must assure that they will be uniformly distributed in the identifier space. Most of the DHTs generate keys for data items directly from their relative values by computing the SHA1 hash of the information provided for storage. Given different inputs, the SHA1 algorithm will produce roughly random 160-bit IDs. 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 uses an XOR metric to measure distances between the leaves of a binary tree, while Chord 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.* [28].

When a participating node wishes to retrieve a non-local value from the system, it issues a lookup query for its corresponding key. The node knows that the value will be stored at some other peer whose ID is closer to the key in question,

thus it propagates the request to another remote node. In order to transfer the lookup to other nodes that fall within the part of the virtual ID space that is closer to the key, each node maintains a routing table which contains network address information for other nodes. As the size of this routing table must be kept to a minimum, each node stores a predefined number of nodes for each part of the network. The distance algorithm plays a major role in overlay segmentation, as each node must store more nodes closer to its own ID and fewer nodes as the distance increases. Moreover, each lookup query may in turn generate another lookup – one step closer to the requested data item. The process will continue, until the value is found or there are no more routing options. Fig. 5 indicates the steps involved in reaching a key-value pair in a common DHT scenario.

Additionally, most systems do not use a recursive process to find a key, but an iteration of lookup commands all coming from one node: The querying node issues a lookup to the peer it knows closest to a value's identifier and waits for a reply containing either data or more routing information. If the node does not get a data reply, it is presented with a list of other peers that are even closer to the key-value pair. It continues querying until it reaches the data item or has no more nodes to propagate requests to. The iterative procedure ensures that messages issued by the querying node also help the peer in finding out which parts of the

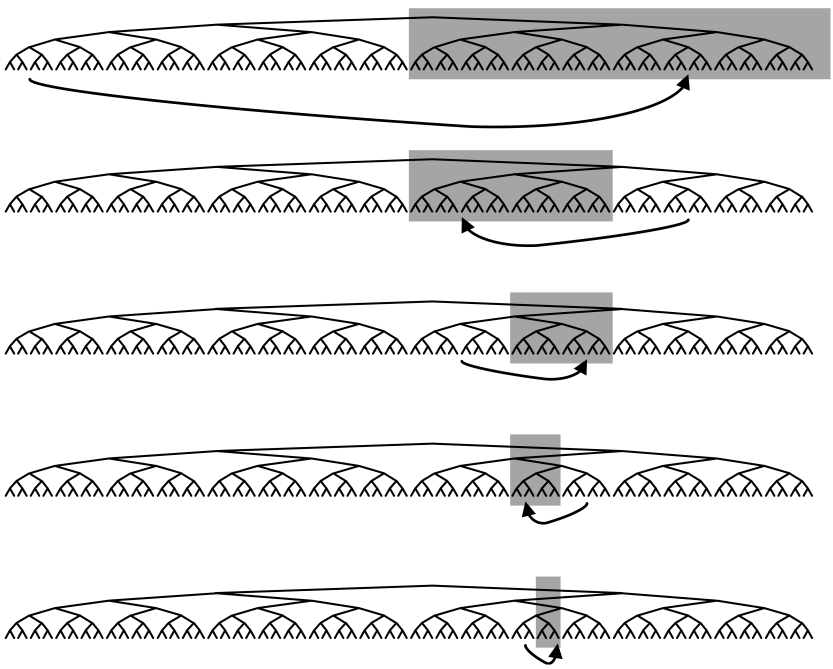

Figure 5: In each query step of a key-value pair lookup in a Distributed Hash Table the node gets one step closer to the requested target.

network are active. Routing tables have to be frequently updated if the overlay network is to remain connected, independent of arbitrary node and network failures. Also, when lookup queries succeed, the results are cached locally to nodes. This way, if a key-value pair is popular, a subsequent query from any part of the network will be satisfied with fewer messages.

An appealing fact is that structured peer-to-peer systems can provide the required mechanisms in order to construct a truly scalable RLS. Actually, the idea of using a peer-to-peer lookup system for locating file replicas in a Grid environment is not new. Foster & Iamnitchi [29] and Iamnitchi *et al.* [30], recognize that the peer-to-peer and Grid research communities have much in common and even more to learn one from another. Furthermore, the authors of the Giggle system credit the work being done in peer-to-peer location discovery systems as most relevant to theirs. In the past few years, peer-to-peer designs have succeeded in drawing the attention of many researchers. Their scalability to very large network sizes and flexibility to random node behavior, have made them a promising solution for the implementation of many distributed systems.

In the context of the Data Grid architecture, the main concern of the RLS 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). LFNs are provided by metadata servers [31] or are hidden in application specific semantics. There is also a strong need that this lookup procedure will complete with the minimum possible latency, 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.

A DHT can be used to support all needs of a Replica Location Service, if its inherent key-value pairs are correlated to LFN to PFN mappings. LFNs can be used as identifiers by the overlay network to route lookup queries to corresponding PFN lists. Furthermore, 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.

In a peer-to-peer RLS, Grid services and end-user applications will access LFN to PFN mappings, by interacting with nodes belonging to the peer-to-peer overlay through predefined APIs. These nodes will practically be applications running on machines connected to the Grid.

5 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 [32]).

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 expire after a predefined interval since their initial publication, and it is the responsibility of systems external to the peer-to-peer network to update or delete them. 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.

5.1 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 [33]. 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

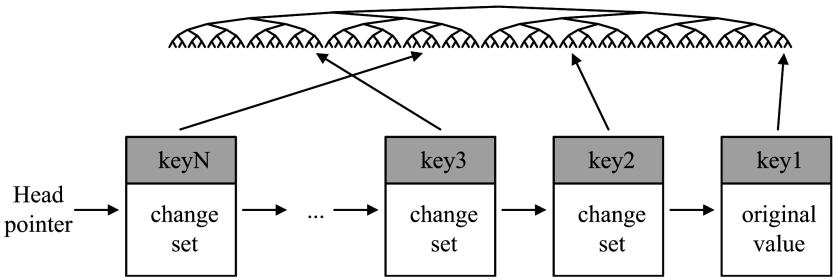

Figure 6: To trace data updates Ivy stores a linked list of read-only key-value pairs in the DHT. However, the identifier of the last change must be kept locally.

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 one or more mappings, 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 [34]. 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 logs are maintained 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 a 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.

5.2 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 storing a specific key-value pair can change over time. DHTs support dynamic node arrivals and departures, so storage relationships between data items and nodes may be altered 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 an 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, until all κ closest peers 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 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_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 latter 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 fig. 7, 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 $\kappa = 2$, a data item will be stored at least at its two closest 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 cannot 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

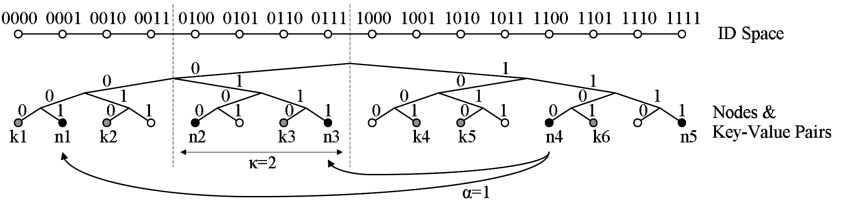

Figure 7: Scaled-down example of a Kademlia network.

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 they expire.

5.3 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 cannot 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 [35]). 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 cannot 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 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. Moreover, there is still a requirement that key-value pairs expire 24 hours after their last modification. Among other advantages refreshing provides is that it is the only way of completely clearing up the ID space of deleted values.

6 Implementation

We implemented the full Kademlia protocol plus our additions in a very light-weight 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 connection-less 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. 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.

6.1 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 $\alpha = 3$ and $\kappa = 4$, as the network size was limited to a few hundred nodes.

Figs 8(a)–(c) 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 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.

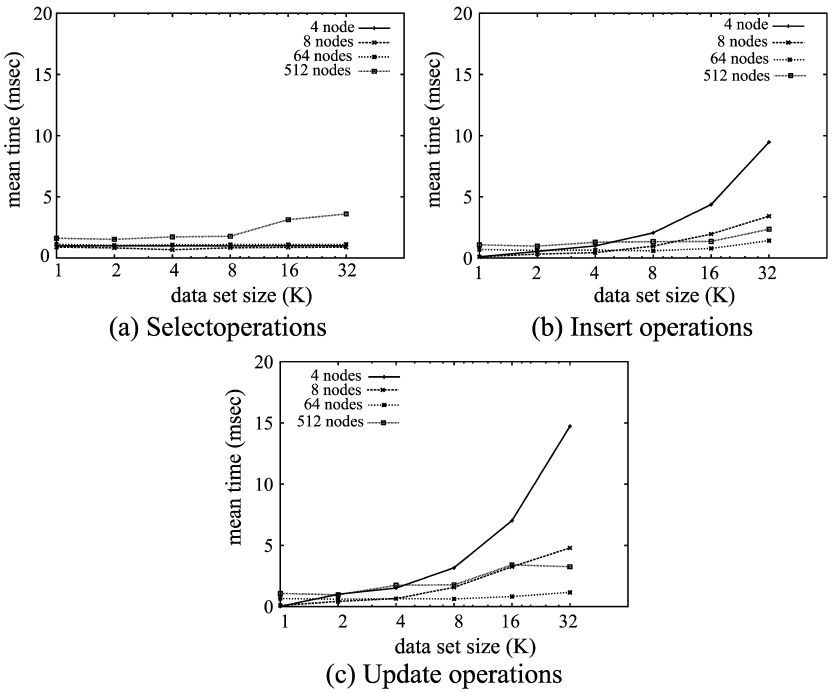

Figure 8: Mean time to complete operations in a static network.

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.

6.2 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 $\alpha = 3$ and $\kappa = 4$, 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. Fig 9 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.

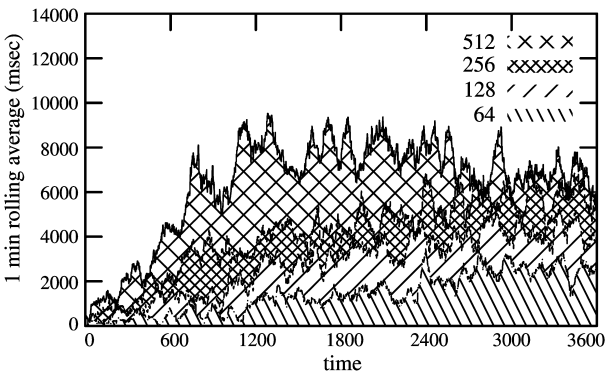

Figure 9: Time to complete queries while increasing node arrival and departure rates.

6.2.1 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.

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 command, 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. Fig. 10 shows a comparison of the two algorithms in a network handling 256 node arrivals and departures per hour.

6.2.2 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

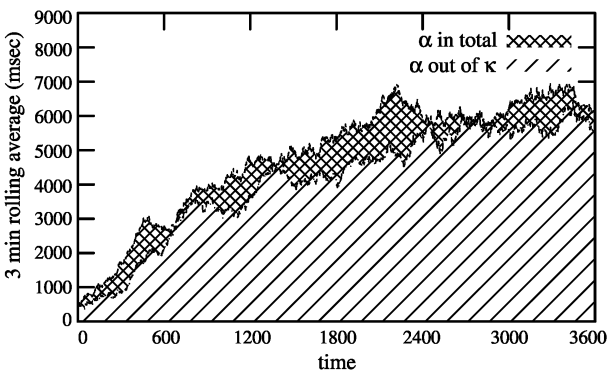

Figure 10: Adapting the α concurrent requests to changes in the top κ entries of the lookup list.

Table 1: Lookup failures while increasing node arrival and departure rates.

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

Table 2: 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%

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.

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.

6.3 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 [36]. 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 [35]. 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 [37] 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.

7 Related work

Peer-to-peer overlay networks and corresponding protocols have already been incorporated in other RLS designs. In a recent paper Min Cai *et al.* [38] 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 [39], 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.

8 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. On 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] Foster, I. & Kesselman, C., (eds.), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan-Kaufmann, 1999.

[2] Foster, I., Kesselman, C. & Tuecke, S., The anatomy of the grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications*, **15(3)**, pp. 200–222, 2001.

[3] Hoschek, W., Jaen-Martinez, J., Samar, A., Stockinger, H. & Stockinger, K., Data management in an international data grid project. *Proceedings of the 1st International Workshop on Grid Computing (Grid 2000)*, Bangalore, India, 2000.

[4] Chervenak, A., Foster, I., Kesselman, C., Salisbury, C. & Tuecke, S., The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, **23(3)**, pp. 187–200, 2000.

[5] Foster, I. & Kesselman, C., Globus: A metacomputing infrastructure toolkit. *International Journal of High Performance Computing Applications*, **11(2)**, pp. 115–128, 1997.

[6] 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. *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press, pp. 1–17, 2002.

[7] Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M. & Werthimer, D., Seti@home: An experiment in public-resource computing. *Communications of the ACM*, **45(11)**, pp. 56–61, 2002.

[8] Zagrovic, B., Snow, C.D., Shirts, M.R. & Pande, V.S., Simulation of folding of a small alpha-helical protein in atomistic detail using worldwide-distributed computing. *Journal of Molecular Biology*, **323(5)**, pp. 927–937, 2002.

[9] The lhc computing grid project (lcg). Http://lcg.web.cern.ch/LCG/.

[10] Vazhkudai, S., Tuecke, S. & Foster, I., Replica selection in the globus data grid. *Proceedings of the International Workshop on Data Models and Databases on Clusters and the Grid (DataGrid 2001)*, Brisbane, Australia, 2001.

[11] Ranganathan, K. & Foster, I., Identifying dynamic replication strategies for a high-performance data grid. *Proceedings of the 2nd International Workshop on Grid Computing (Grid 2001)*, Denver, CO, 2001.
[12] Kunszt, P.Z., Laure, E., Stockinger, H. & Stockinger, K., Advanced replica management with reptor. *Proceedings of the 5th International Conference on Parallel Processing and Applied Mathematics (PPAM 2003)*, Springer Verlag, pp. 848–855, 2003.
[13] Guy, L., Kunszt, P., Laure, E., Stockinger, H. & Stockinger, K., Replica management in data grids. Technical report, GGF5 Working Draft, July 2002.
[14] The datagrid project. Http://www.eu-datagrid.org/.
[15] Enabling grids for e-science in Europe (egee). Http://www.eu-egee.org/.
[16] Bosio, D., *et al.,* Next-generation EU datagrid data management services. *Proceedings of the 2003 conference on Computing in High Energy and Nuclear Physics (CHEP03)*, San Diego, CA, 2003.
[17] Dullmann, D., Hoschek, W., Jean-Martinez, J., Samar, A., Stockinger, H. & Stockinger, K., Models for replica synchronisation and consistency in a data grid. *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, San Francisco, CA, 2001.
[18] Samar, A. & Stockinger, H., Grid data management pilot (gdmp): A tool for wide area replication. *Proceedings of IASTED International Conference on Applied Informatics (AI2001)*, Innsbruck, Austria, 2001.
[19] Stockinger, H., Samar, A., Holtman, K., Allcock, B., Foster, I. & Tierney, B., File and object replication in data grids. *Cluster Computing*, **5(3)**, pp. 305–314, 2002.
[20] Stockinger, H. & Hanushevsky, A., Http redirection for replica catalogue lookups in data grids. *Proceedings of the 17th ACM Symposium on Applied Computing (SAC2002)*, Madrid, Spain, 2002.
[21] Chervenak, A., Palavalli, N., Bharathi, S., Kesselman, C. & Schwartzkopf, R., Performance and scalability of a replica location service. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13'04)*, Honolulu, HI, 2004.
[22] Ripeanu, M. & Foster, I., Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, 2002.
[23] Adamic, L.A., Lukose, R.M., Puniyani, A.R. & Huberman, B.A., Search in power law networks. *Physical Review E64*, **64**, pp. 46135–46143, 2001.
[24] Maymounkov, P. & Mazières, D., Kademlia: A peer-to-peer information system based on the xor metric. *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, 2002.
[25] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F. & Balakrishnan, H., Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM Press, pp. 149–160, 2001.

[26] 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(1)**, pp. 41–53, 2004.

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

[28] Balakrishnan, H., Kaashoek, M.F., Karger, D., Morris, R. & Stoica, I., Looking up data in p2p systems. *Communications of the ACM*, **46(2)**, pp. 43–48, 2003.

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

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

[31] Czajkowski, K., Fitzgerald, S., Foster, I. & Kesselman, C., Grid information services for distributed resource sharing. *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01)*, IEEE Computer Society Press, p. 181, 2001.

[32] Hazel, S. & Wiley, B., Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems. *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, 2002.

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

[34] 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. *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge, MA, 2000.

[35] Li, J., Stribling, J., Gil, T.M., Morris, R. & Kaashoek, M.F., Comparing the performance of distributed hash tables under churn. *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, San Diego, CA, 2004.

[36] Buchmann, E. & Böhm, K., How to run experiments with large peer-to-peer data structures. *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Santa Fe, NM, 2004.

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

[38] Cai, M., Chervenak, A. & Frank, M., A peer-to-peer replica location service based on a distributed hash table. *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Pittsburgh, PA, 2004.
[39] Ripeanu, M. & Foster, I., A decentralized, adaptive, replica location service. *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11'02)*, Edinburgh, UK, 2002.