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
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.

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 [11, 12]. A critical component of this infrastructure, commonly referred to as Grid "middleware", is the data management layer. Pioneering Grid efforts [14, 27] 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 [6] allows the distributed storage and management of 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 layers.

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 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 are called "replicas" [24], while applications running on the Grid request such local file instances through specialized Grid data management services [17]. 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.

Replica Location Service implementations have evolved significantly over the past years. The initial design of a centralized RLS was swiftly put aside in favor of a distributed approach [23]. The most widespread solution currently deployed on the Grid, namely the Giggle Framework [5], follows a multi-tier hierarchical structure, distributing data and queries over global (Grid or VO-wide) and local, site-wide RLS instances. Giggle is currently an integral component of the Globus Toolkit [10] middleware distribution. Nevertheless, all implementations followed so far are optimized for "high-performance" operational environments. Current Grid deployments reside mainly in the scientific area, where hardware crashes and network blackouts are rare exceptions and may be sustained by redundant equipment or special

backup systems. However, in future global-scale, "high-throughput" Grids, services like the RLS may have to address these issues. 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. Therefore, service redesigns may benefit from concepts and algorithms used by peer-to-peer overlay networks. Peer-to-peer systems can scale without application and environment specific fine-tuning to billions of simultaneous participants, while their potential grows as more peers join in.

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 a special category of peer-to-peer 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 analyze the complications associated with supporting update operations in DHTs 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 DHT protocol, discuss on the method and evaluate its merits, based 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.

2 Limitations of the Giggle Framework

As most other preceding RLS designs, the Giggle (GIGa-scale Global Location Engine) Framework 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, while PFNs, which are used by the RLS 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). RLIs help in finding which catalogs hold the replica file lists for a given LFN, while LRCs maintain the actual replica location information (LFN to PFN mappings). To meet varying operational requirements, multiple RLIs may be deployed in parallel, providing optional coarse-grain load-balancing and fail-over features to the replica location infrastructure. A standard deployment scenario may include running one LRC per site and a multitude of VO-wide RLIs in a tree-like structure. Each LRC may be linked to multiple RLIs and vice versa. The exact form of the catalog and index hierarchy can be controlled through the definition of a number of deployment parameters. Nevertheless, changes in any LFN's replicas, will all be concentrated at the LRCs responsible for storing the particular mappings. Specific catalogs may get overloaded when very popular LFNs require frequent updates of their associated PFN lists.

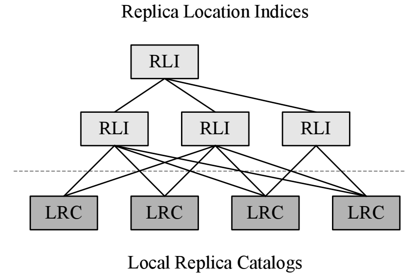


Figure 1. Giggle deployment example

LRCs are required to refresh RLIs, in order to inform them on the latest mapping updates and to prevent them from deleting old mappings because of timeouts. As the update mechanism has to be as efficient as possible, LRCs use soft-state update protocols to inform RLIs of changes. Either full or incremental, updates are asynchronous, so when an add or delete operation occurs, it is not immediately propagated to the appropriate index. 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. Asynchronous updates and lossy compression of data, may result in clients getting false positive answers. Although the relaxed consistency requirements set by Giggle's designers allow false positives, we believe that the pursuit of scalability has led Giggle to employ complex mechanisms to update data which may in turn limit the system's efficiency on very large networks.

There is also an option to partition mappings among global servers, by defining an LFN namespace segmentation function. Data partitioning can be used to limit the amount of changes that need to be communicated between LRCs and RLIs, but it has to be manually configured. 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 a new distribution scheme. In general, Giggle's parameters cannot be dynamically changed.

According to the experimental analysis of a prototype implementation [7], 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. Moreover, the catalogs and indices cannot automatically handle a new addition or deletion of a participating catalog or index. 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 implies that every time a new entity is to be added in the network the whole service may have to be reconfigured.

The number of parameters that have to be tuned in order to deploy and use the RLS, make Giggle difficult to deploy and manage. Moreover, 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. 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. Nevertheless, we propose that a Replica Location Service for high-throughput Grid deployments, can be implemented with the help of an already scalable, fault-tolerant and self-configurable peer-to-peer network.

3 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 location services.

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. Unstructured systems like Gnutella leave the peers free to participate in any part of the overlay and the connection graph formed resembles that of a power-law network [22], while structured systems or Distributed Hash Tables (DHTs), such as Kademlia [19], Chord [25] and Tapestry [26], impose a specific virtual structure which accommodates peers in particular slots as they join the network. 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 the logarithm of the number of total participating nodes). On the other hand, unstructured systems, have the ability to handle free-text search queries in very few steps [1], although the procedure is probabilistic and usually requires flooding the network with messages.

An appealing fact is that structured peer-to-peer systems can provide the required mechanisms in order to construct a truly scalable RLS. DHTs 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 [2]. Actually, 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, 15], 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.

3.1 Design

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. LFNs are provided by metadata servers [8] or are hidden in application specific semantics. 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. In such a system, keys will not be generated by computing the SHA1 hash of the value (as is the common case with DHTs). Keys should correspond to the hash of the logical filename (LFN) of the resource. LFN hashes can then be used as identifiers by the overlay network to route data operations to corresponding PFN lists. A value for a key will actually be a data structure - a list containing the physical locations of replicas (PFNs) for a given LFN. Also, as LFNs are unique per VO identifiers, a single peer-to-peer overlay network must be deployed per VO (a single identifier space). Grid services and end-user applications will access LFN to PFN mappings, by interacting with applications participating in the peer-to-peer overlay, through predefined APIs.

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. These systems are tuned to scale to very large network sizes and adapt to random node behavior. Tracing the nodes responsible for storing a specific data item would require complex and demanding algorithms, so there is no method to determine the exact location(s) of a key-value tuple in any given moment (this is also a prerequisite for peer-to-peer network security [13]). Nevertheless, a data *update* operation is absolutely necessary for serving the needs of the Data Grid's RLS, 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.

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 DHT 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.

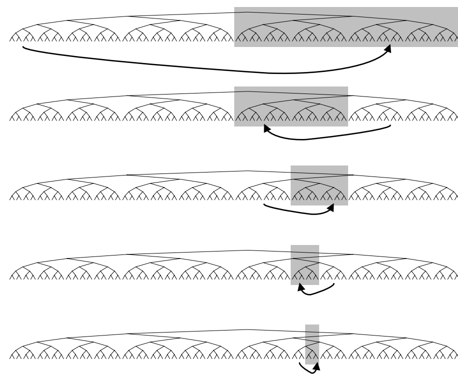


Figure 2. Moving closer towards a key in a Kademlia overlay

3.2 Enabling mutable data storage

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. 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 (Figure 6).

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. When all κ closest nodes have replied, the key-value pair is copied to the corresponding peers via `STORE` RPCs. 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. 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 ei-

ther 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.

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, so there is no need to send `FIND_VALUE` RPCs to the closest nodes of a key (version checking is done by the remote peers). Moreover, deleting a value equals to updating it to zero length. Deleted data will eventually be removed from the system when it expires.

3.3 Discussion

In the original Kademlia protocol, a *lookup* operation will normally require at most log(N) hops through a network of N peers. 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 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 peer-to-peer systems [18]). Moreover, the aforementioned drawback in lookup performance could even be accounted as a feature: DHT nodes generally exploit messages exchanged in favor of updating their routing tables. The more the messages, the more fault-tolerant the system gets.

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. Also, as values are automatically republished on every usage, they is no need to explicitly redistribute key-value pairs every hour. Data items are reseeded only when an hour passes since they were last part of a *store* or *lookup* operation (effectively propagating updates). Nevertheless, there is still a requirement that all tuples 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.

4 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.

4.1 Performance in a static network

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

Figures 3(a), 3(b) and 3(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 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.

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.

4.2 Performance in a dynamic network

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

Each experiment involved node arrivals and departures, as long as item lookups and updates, during a one hour timeframe. Corresponding *startup*, *shutdown*, *get* and *set* commands were generated randomly according to a Poisson distribution, and then issued in parallel to the nodes. We started by setting the item update and lookup rates to 1024 $\frac{operations}{hour}$, while doubling the node arrival and departure rates. Initially 64 new nodes were generated per hour and 64 $\frac{nodes}{hour}$ failed. The arrival and departure rates were kept equal so that the network would neither grow nor shrink. Figure 4 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.

4.2.1 Handling timeouts

As expected, increasing the number of node failures, caused the total time needed for the completion of each query to

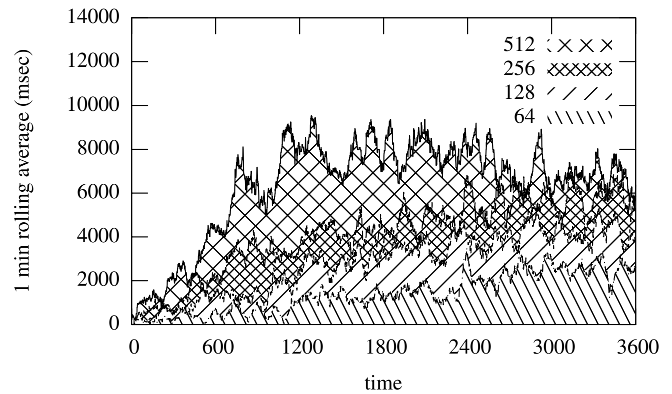


Figure 4. Time to complete queries while increasing node arrival and departure rates

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.

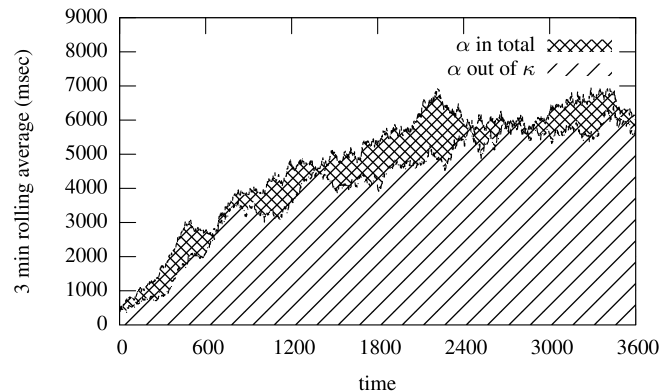
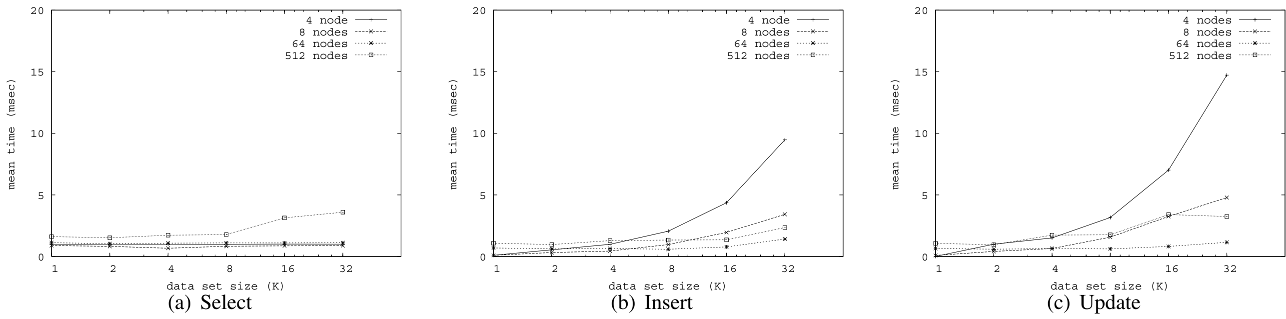


Figure 5. Dynamically adapting α to changes in the lookup list

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 com-



(a) Select (b) Insert (c) Update

Figure 3. Mean time to complete operations in a static network

mands, 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 5 shows a comparison of the two algorithms in a network handling 256 node arrivals and departures per hour.

4.2.2 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 node failures per hour) several times, while doubling the lookup rate from 1024 up to 16384 $\frac{operations}{hour}$. It is evident from the results presented in Table 2, that even in a network with very unreliable peers, a high lookup rate can cause the corresponding failure rate to drop to values less than 1%. This owes to the fact that lookup operations are responsible for propagating key-value pairs to a continuously changing set of closest nodes, while helping peers find and remove stale entries from their routing tables.

Table 1. Increasing node node arrival and departure rates

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

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

Table 2. Increasing the lookup rate

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

4.3 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. We will be focusing on evaluating various aspects of the system, while varying node network characteristics. 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 [3].

In future versions of the implementation, we intend on evaluating embedded, lightweight database engines like SQLite for the local storage requirements of each node.

We also plan on adding support for Kademlia's *accelerated lookups* and integrate interfaces to the advanced security services provided by Grid middleware distributions.

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.

5 Related work

A solution to the problem of storing mutable data in a DHT is presented by the designers of Ivy [20]. Ivy is a distributed filesystem 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 filesystem 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 filesystem can "remember" the latest state of each file and directory for a given participant. Nevertheless, there may be a need to go through hundreds of key-value lookups in the DHT in order to find the latest aggregate value, which would incur an intolerable cost in terms of network messages. Even more, 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 [16], which implements a file management layer on top of an underlying Tapestry network. The update model used is very similar to the one utilized by Ivy, although updates are handled at the file - not the participant - level. If any of these systems was to be used as the basis for an RLS, there would be a need to maintain an external mutable directory of the latest keys inserted in the peer-to-peer overlay.

Peer-to-peer overlay networks have already been incorporated in other RLS designs. In a recent paper [4], 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.

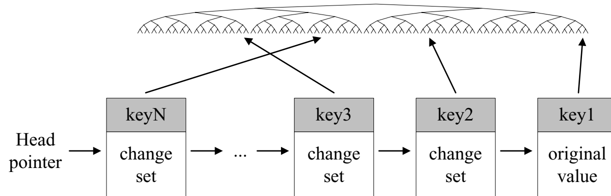


Figure 6. Storing a mutable log of read-only key-value pairs in a DHT

In another variant of an RLS implementation using a peer-to-peer network [21], 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.

6 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 plan 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. In contrast to Giggle and other peer-to-peer RLS designs, we envision a service that does not require the use of specialized servers for locating replicas. We believe that a lightweight DHT-enabled RLS peer can run at multiple machines per site or even every machine of the Grid having a public IP address, as the deployment and management 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.

References

[1] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman. Search in power law networks. *Physical Review E64*, 64:46135–46143, 2001.

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

[3] E. Buchmann and K. Böhm. How to Run Experiments with Large Peer-to-Peer Data Structures. In *Proc. of IPDPS'04*, Santa Fe, NM.

[4] M. Cai, A. Chervenak, and M. Frank. A peer-to-peer replica location service based on a distributed hash table. In *Proc. of ACM/IEEE Supercomputing'04*, Pittsburgh, PA.

[5] A. Chervenak et al. Giggle: a framework for constructing scalable replica location services. In *Proc. of ACM/IEEE Supercomputing'02*, Baltimore, MD.

[6] 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*, 23(3):187–200, 2000.

[7] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and scalability of a replica location service. In *Proc. of HPDC–13'04*, Honolulu, HI.

[8] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proc. of HPDC–10'01*, San Francisco, CA.

[9] I. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Proc. of IPTPS'03*, Berkeley, CA.

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

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

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

[13] S. Hazel and B. Wiley. Achord: A variant of the chord lookup service for use in censorship resistant peer-to-peer publishing systems. In *Proc. of IPTPS'02*, Cambridge, MA.

[14] W. Hoschek, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data grid project. In *Proc. of Grid'00*, Bangalore, India.

[15] A. Iamnitchi, I. Foster, and D. C. Nurmi. A peer-to-peer approach to resource location in grid environments. In *Proc. of HPDC–11'02*, Edinburgh, UK.

[16] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS–IX'00*, Cambridge, MA.

[17] P. Z. Kunszt, E. Laure, H. Stockinger, and K. Stockinger. Advanced replica management with reptor. In *Proc. of PPAM'03*, Czestochowa, Poland.

[18] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. of IPTPS'04*, San Diego, CA.

[19] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of IPTPS'02*, Cambridge, MA.

[20] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. of OSDI'02*, Boston, MA.

[21] M. Ripeanu and I. Foster. A decentralized, adaptive, replica location service. In *Proc. of HPDC–11'02*, Edinburgh, UK.

[22] M. Ripeanu and I. Foster. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. In *Proc. of IPTPS'02*, Cambridge, MA.

[23] H. Stockinger and A. Hanushevsky. Http redirection for replica catalogue lookups in data grids. In *Proc. of SAC'02*, Madrid, Spain.

[24] H. Stockinger, A. Samar, K. Holtman, B. Allcock, I. Foster, and B. Tierney. File and object replication in data grids. *Cluster Computing*, 5(3):305–314, 2002.

[25] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of SIGCOMM'01*, San Diego, CA.

[26] 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*, 22(1):41–53, 2004.

[27] The datagrid project. http://www.eu-datagrid.org/.