A Peer-to-Peer Replica Location Service Based on A Distributed Hash Table

Min Cai
USC Information Sciences Institute
4676 Admiralty Way,
Marina del Rey, CA 90292
mcai@isi.edu

Ann Chervenak
USC Information Sciences Institute
4676 Admiralty Way,
Marina del Rey, CA 90292
annc@isi.edu

Martin Frank
USC Information Sciences Institute
4676 Admiralty Way,
Marina del Rey, CA 90292
frank@isi.edu

ABSTRACT

A Replica Location Service (RLS) allows registration and discovery of data replicas. In earlier work, we proposed an RLS framework and described the performance and scalability of an RLS implementation in Globus Toolkit Version 3.0. In this paper, we present a Peer-to-Peer Replica Location Service (P-RLS) with properties of self-organization, fault-tolerance and improved scalability. P-RLS uses the Chord algorithm to self-organize P-RLS servers and exploits the Chord overlay network to replicate P-RLS mappings adaptively. Our performance measurements demonstrate that update and query latencies increase at a logarithmic rate with the size of the P-RLS network, while the overhead of maintaining the P-RLS network is reasonable. Our simulation results for adaptive replication demonstrate that as the number of replicas per mapping increases, the mappings are more evenly distributed among P-RLS nodes. We introduce a predecessor replication scheme and show it reduces query hotspots of popular mappings by distributing queries among nodes.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Systems**]: Distributed Systems; C.4 [**Performance of Systems**]: Design Studies

General Terms

Algorithms, Experimentation

Keywords

Grid, Peer-to-Peer, Replication

1. INTRODUCTION

In Grid environments, replication of remote data is important for data intensive applications. Replication is used for fault tolerance as well as to provide load balancing by allowing access to multiple replicas of data. One component of a scalable and reliable replication management system is a Replica Location Service (RLS) that allows the registration and discovery of replicas. Given a logical identifier of a data object, the Replica Location Service must provide the physical locations of the replicas for the object. In earlier work, Chervenak et al. [1] proposed a parameterized RLS framework that allows users to deploy a range of replica location services that make tradeoffs with respect to consistency, space overhead, reliability, update costs, and query costs by varying six system design parameters. A Replica Location Service implementation based on this framework is currently available as part of the Globus Toolkit Version 3 [2].

The Replica Location Service design consists of two components. Local Replica Catalogs (LRCs) maintain consistent information about logical-to-physical mappings on a site or storage system, and Replica Location Indices (RLIs) aggregate information about mappings contained in one or more LRCs. The RLS achieves reliability and load balancing by deploying multiple and possibly redundant RLIs in a hierarchical, distributed index. An example RLS deployment is shown in Figure 1.

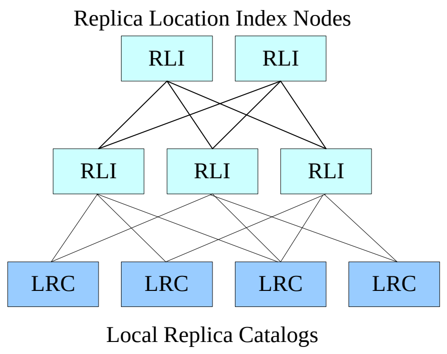

Figure 1: Example of a hierarchical RLI Index configuration supported by the RLS implementation available in the Globus Toolkit Version 3.

The RLS framework also envisions a membership management service that keeps track of LRCs and RLIs as they enter and leave the system and adapts the distributed RLI index according to the current server membership. However, the current RLS implementation does not contain a membership service; instead, it uses a static configuration of LRCs and RLIs that must be known to servers and clients.

The current RLS implementation is being used successfully in production mode for several scientific projects, including as the Earth System Grid [3] and the Laser Interferometer Gravitational Wave Observatory [4]. However, there are several features of the existing RLS that could be improved. First, because a membership service has not been implemented for the RLS, each deployment is statically configured, and the system does not automatically react to changes in membership (i.e., servers joining or leaving the system). Configuration files at each RLS server specify parameters including authorization policies; whether a particular server will act as an LRC, an RLI or both; how state updates are propagated from LRCs to RLIs; etc. When new servers are added to or removed from the distributed RLS system, affected configuration files are typically updated via command-line administration tools to reflect these changes. While this static configuration scheme has proven adequate for the scale of current deployments, which typically contain fewer than ten RLS servers, more automated and flexible membership management is desirable for larger deployments. Second, although the current RLS provides some fault tolerance by allowing LRCs to send state updates to more than one RLI index node, the overall RLI deployment is statically configured and does not automatically recover from RLI failures. We have no ability to specify, for example, that we want the system to maintain at least 3 copies of every mapping in the RLI index space or that after an RLI server failure, the distributed RLS should automatically reconfigure its remaining servers to maintain the required level of redundancy.

The goal of our current work is to use a peer-to-peer approach to provide a distributed RLI index with properties of self-organization, greater fault-tolerance and improved scalability. We have designed a Peer-to-Peer Replica Location Service (P-RLS) design that uses the overlay network of the Chord peer-to-peer system [5] to self-organize P-RLS servers. A P-RLS server consists of an unchanged Local Replica Catalog (LRC) and a peer-to-peer Replica Location Index node called a P-RLI. The network of P-RLIs uses the Chord routing algorithm to store mappings from logical names to LRC sites. A P-RLI responds to queries regarding the mappings it contains and routes queries for other mappings to the P-RLI nodes that contain those mappings. The P-RLS system also exploits the structured overlay network among P-RLI nodes to replicate mappings adaptively among the nodes; this replication of mappings provides a high level of reliability and availability in the P-RLS network.

We implemented a prototype of the P-RLS system by extending the RLS implementation in Globus Toolkit Version 3.0 with Chord protocols. We evaluated the performance and scalability of a P-RLS network with up to 16 nodes containing 100,000 or 1 million total mappings. We also simulated the distribution of mappings and queries in P-RLS systems ranging in size from 10 to 10,000 nodes that contain a total of 500,000 replica mappings. In this paper, we describe the P-RLS design, its implementation and our performance results.

2. BACKGROUND

2.1 The Globus Toolkit Replica Location Service Implementation

The RLS included in the Globus Toolkit Version 3 provides a distributed registry that maintains information about physical locations of copies and allows discovery of replicas. The RLS framework [1] consists of five components: the LRC, the RLI, a soft state maintenance mechanism, optional compression of state updates, and a membership service. The LRCs maintain mappings between logical names of data items and target physical names. The RLIs aggregate state information contained in one or more LRCs and build a hierarchical, distributed index to support discovery of replicas at multiple sites, as shown in Figure 1. LRCs send summaries of their state to RLIs using soft state update protocols. Information in RLIs times out and must be periodically refreshed. To reduce the network traffic of soft state updates and RLI storage overheads, the RLS also implements an optional Bloom filter compression scheme [6]. In this scheme, each LRC only sends a bit map that summarizes its mappings to the RLIs. The bit map is constructed by performing a series of hash functions on logical names that are registered in an LRC and setting the corresponding bits in the bit map. Bloom filter compression greatly reduces the overhead of soft state updates. However, the bloom filter is a lossy compression scheme. Using bloom filters, the RLIs lose information about specific logical names registered in the LRCs. There is also a small probability that the Bloom filter will provide a false positive, an incorrect indication that a mapping exists in the corresponding LRC when it does not. A membership service is intended to keep track of participating LRCs and RLIs as well as which servers send and receive soft state updates from one another. The current implementation does not include a membership service but rather maintains a static configuration for the RLS.

The RLS is implemented in C and uses the *globus_io* socket layer from the Globus Toolkit. The server consists of a multi-threaded front end server and a back-end relational database, such as MySQL or PostgreSQL. The front end server can be configured to act as an LRC server and/or an RLI server. Clients access the server via a simple string-based RPC protocol. The client APIs support both C, Java and Python. The implementation supports two types of soft state updates from LRCs to RLIs: (1) a complete list of logical names registered in the LRC and (2) Bloom filter summaries of the contents of an LRC. The implementation also supports partitioning of the soft state updates based on pattern matching of logical names.

The distributed RLI index can provide redundancy and/or partitioning of the index space among RLI index nodes. LRCs can be configured to send soft state updates summarizing their contents to one or more RLIs. When these updates are sent to multiple RLIs, we avoid having performance bottlenecks or single points of failure in the index space. In the framework design as well as the Globus Toolkit 3 implementation, RLS also supports the capability of limiting the size of soft state updates based on a partitioning of the logical namespace. With partitioning, we perform pattern matching of logical names and send only matching updates to a specified RLI index. The concept of partitioning was considered important to reduce the network and memory requirements for sending soft state updates. In practice, however, the use of Bloom filter compression is so efficient at reducing the size of updates that partitioning is rarely used.

While the current implementation of the RLS is being used effectively in several Grid production deployments and systems [3, 7-9], we are interested in applying peer-to-peer ideas to the distributed RLI index to produce an index that is self-configurable, highly fault tolerant and scalable.

2.2 Peer to Peer Systems and Chord

Peer-to-peer (P2P) systems can be categorized as either unstructured or structured networks. These systems provide failure tolerant approaches to looking up the location of an object. The Gnutella [10] peer-to-peer file sharing system uses an unstructured network among peers; each query for an object location is flooded to the whole network. However, measurement studies show that this approach does not scale well because of the large volume of query messages generated by flooding [11-13]. By contrast, structured P2P networks such as those using distributed hash tables (DHTs) [14] maintain a structured overlay network among peers and use message routing instead of flooding. The basic functionality they offer is *lookup (key)*, which returns the identity of the node storing the object with that key. Recent proposed DHT systems include Tapestry [15], Pastry [16], Chord [5], CAN [17] and Koorde [18]. In these DHT systems, objects are associated with a key that can be produced by hashing the object name. Nodes have identifiers that share the same space as keys. Each node is responsible for storing a range of keys and corresponding objects. The DHT nodes maintain an overlay network, with each node having several other nodes as neighbors. When a *lookup (key)* request is issued from one node, the lookup message is routed through the overlay network to the node responsible for the key. Different DHT systems construct a variety of overlay networks and employ different routing algorithms. They can guarantee to finish a lookup operation in $O(\log N)$ or $O(dN^{1/d})$ hops, and each node only maintains the information of $O(\log N)$ or d neighbors for an N node network (where d is the dimension of the hypercube organization of the network). Therefore, these DHT systems provide good scalability as well as failure resilience.

Our design of the P-RLS is based on the Chord system. Next, we briefly describe the basic Chord algorithm proposed by Stoica, et al. [5]. Chord uses a one-dimensional circular identifier space with modulo 2^m for both node identifiers and object keys. Every node in Chord is assigned a unique m-bit identifier by hashing their IP address and port number, and all nodes self-organize into a ring topology based on their node identifiers in the circular space. Each object is also assigned a unique m-bit identifier called its *object key*. Object keys are assigned to nodes by using *consistent hashing*, i.e., key k is assigned to the first node whose identifier is equal to or follows the identifier of k in the circular space. This node is responsible for storing the object with key k and is called its *successor node*, denoted by *successor(k)*.

Each Chord node maintains two sets of neighbors, its successors and its fingers. The successor nodes immediately follow the node in the identifier space, while the finger nodes are spaced exponentially around the identifier space. Each node has a constant number of successors and at most m fingers. The i-th finger for the node with identity n is the first node that succeeds n by at least 2^{i-1} on the identifier circle, where $1 \le i \le m$. The first finger node is the immediate successor of n, where i=1. When node n wants to lookup the object with key k, it will route a lookup request to the successor node of key k. If the successor node is far away from n, node n forwards the request to the finger node whose identifier most immediately precedes the successor node of key k. By repeating this process, the request gets closer and closer to the successor node. Eventually, the successor node receives the lookup request for the object with key k, finds the object locally and sends the result back to node n. Because the fingers of each node are spaced exponentially around the identifier space, each hop from one node to the next node covers at least half the identifier space (clockwise) between that node and the successor node of key k. So the number of routing hops for a lookup is $O(\log N)$ for a Chord network with N nodes. In addition, each node only needs to maintain pointers to $O(\log N)$ neighbors.

Chord achieves load balancing of nodes by using consistent hashing and virtual nodes. Consistent hashing assigns each object key to the first node whose identifier is equal to or follows the object key in the circular space, so the number of keys stored on each node is determined by the distance of the node to its immediate predecessor in the circular space. However, the node identifiers generated by SHA1 hashing do not uniformly cover the entire space. Chord solves this problem by associating object keys to virtual nodes, and mapping multiple virtual nodes to each real node. Each virtual node has its own node identifier in the circular space and maintains the separated neighborhood information of other virtual nodes.

To maintain the ring topology correctly when nodes join and leave, each Chord node also runs a stabilization protocol periodically in the background that ensures each node's successor pointer is up to date and improves the finger table for better lookup performance. Chord achieves fault tolerance for its ring topology and routing by maintaining a constant number of successors for each node. However, Chord does not provide fault tolerance for the data stored on its nodes; this data maybe be lost when a node fails. Section 3.1 discusses our approach to providing greater fault tolerance by adaptively replicating mappings on multiple P-RLS nodes. Our scheme leverages the membership information provided by Chord to perform this adaptive replication.

3. THE PEER-TO-PEER REPLICA LOCATION SERVICE DESIGN

Next, we describe the design of our peer-to-peer Replica Location Service (P-RLS). This design replaces the hierarchical RLI index from the Globus Toolkit Version 3 RLS implementation with a self-organizing, peer-to-peer network of P-RLS nodes.

In the P-RLS system, the Local Replica Catalogs (LRCs) are unchanged. Each LRC has a local P-RLI server associated with it, and each P-RLI node is assigned a unique m-bit Chord identifier. The P-RLI nodes self-organize into a ring topology based on the Chord overlay construction algorithm discussed in Section 2.2. The P-RLI nodes maintain connections to a small number of other P-RLI nodes that are their successor nodes and finger nodes. When P-RLI nodes join or leave, the network topology is repaired by running the Chord stabilization algorithm. Thus, the Chord overlay network provides membership maintenance for the P-RLS system.

Updates to the Replica Location Service begin at the Local Replica Catalog (LRC), where a user registers or unregisters replica mappings from logical names to physical locations. LRCs periodically send soft state updates summarizing their state into the peer-to-peer P-RLS network. The soft state update implementation in P-RLS is based on the uncompressed soft state updates of the original RLS implementation. Just as in that implementation, our updates contain {*logical name, LRC*} mappings. To perform a soft state update in P-RLS, the system first generates the Chord key identifier for each logical name in the soft state update by applying an SHA1 hash function to the logical names. Then the system identifies the P-RLI successor node of the Chord key of each logical name and stores the corresponding {*logical name, LRC*} mapping on that node. We call this successor node the *root node* of the mapping. Figure 2 shows how three mapping are placed in a P-RLS network with 8 nodes.

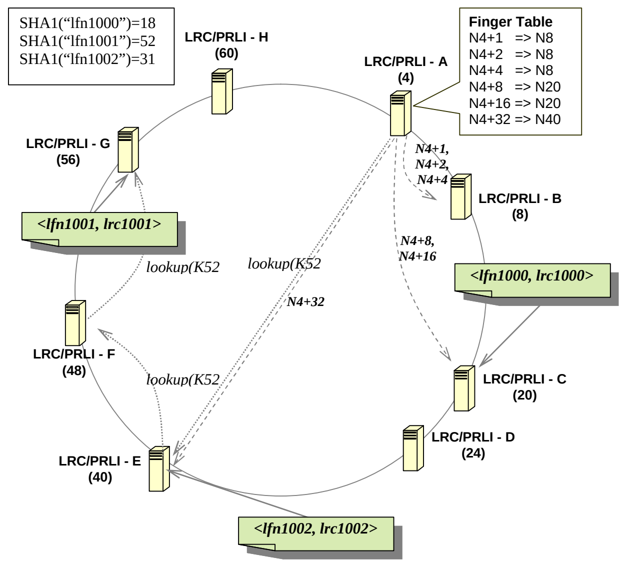

Figure 2: Example of the mapping placement of 3 mappings in the P-RLS network with 8 nodes.

To locate an object in the P-RLS system, clients can submit queries to any P-RLS node. When a P-RLS node receives a query for a particular logical name, it generates the Chord key for that name and checks whether it is the successor node for that key. If so, then this node contains the desired {*logical name, LRC*}; the node searches its local RLI database and returns the query result to the client. Otherwise, the node will determine the successor node for the object using the Chord successor routing algorithm and will forward the client's query to the successor node, which returns zero or more {*logical name, LRC*} mappings to the client. Once the the client receives these P-RLS query results, the client makes a separate query to one or more LRCs to retrieve mappings from the logical name to one or more physical locations of replicas. Finally, the client can access the physical replica.

Next, we describe several aspects of our P-RLS design, including adaptive replication of P-RLI mappings and load balancing.

3.1 Adaptive Replication

The P-RLI nodes in the P-RLS network can join and leave at any time, and also the network connection between any two nodes can be broken. In order to resolve queries for {*logical name, LRC*} mappings continuously despite node failures, we need to replicate the mappings on different P-RLI nodes. In the P-RLS network, the Chord membership maintenance protocol can maintain the ring topology among P-RLI nodes even when a number of nodes join and leave concurrently. Thus, it is quite intuitive to replicate our mappings in the P-RLS network based on the membership information provided by the Chord protocol.

Based on the above P-RLS design, we know that each mapping will be stored on the root node of the mapping. The root node maintains the connections to its k successor nodes in the Chord ring for successor routing reliability, where k is the *replication factor* and is typically $O(\log N)$ for a P-RLS network with N nodes. Thus, the total number of copies of each mapping is $k+1$.

A simple replication approach is to replicate the mappings stored on the root node to its k successors. This scheme, called *successor replication,* is adaptive when nodes join or leave the system. When a node joins the P-RLS network, it will take over some of the mappings and replicas from its successor node. When a node leaves the system, no explicit handover procedure is required, and the node does not need to notify its neighbors; the Chord protocol running on the node's predecessor will detect its departure, make another node the new successor, and replicate mappings on the new successor node adaptively. If, because of membership changes in the P-RLS network, a particular node is no longer a successor of a root node, then the mappings from that root node need to be removed from the former successor node. We solve this problem by leveraging the soft state replication and the periodic probing messages of the Chord protocol. Each mapping has an expiration time, and whenever a node receives a probe message from its predecessor, it will extend the expiration time of the mappings belonging to that predecessor, because the node knows that it is still the successor node of that predecessor. Expired mappings are timed out to avoid unnecessary replication of mappings. When a mapping on a root node is updated by an LRC, the root node updates its successors immediately to maintain the consistency of replicated mappings. Since the successor replication scheme adapts to nodes joining and leaving the system, the mappings stored in the P-RLS network will not be lost unless all k successors of a particular root node fail simultaneously.

3.2 Load Balancing

Load balancing is another important problem for a distributed replication index system, such as P-RLS. Here, we consider two aspects of the load balancing problem: evenly distributing mappings among nodes and query load balancing for extremely popular mappings.

3.2.1 Even Distribution of Mappings among Nodes

The Chord algorithm we discussed in section 2.2 uses consistent hashing and virtual nodes to balance the number of keys stored on each node. However, the virtual nodes approach introduces some extra costs, such as maintaining more neighbors per node and increasing the number of hops per lookup.

We adaptively replicate mappings on multiple P-RLS nodes for fault tolerance purpose. At the same time, mapping replication can improve the distribution of mappings among nodes without using virtual nodes. In P-RLS, the number of {*logical name*, LRC} mappings stored on each P-RLI node is determined by the distance of the node to its immediate predecessor in the circular space, i.e. the "*owned region*" of the P-RLI node. In Chord [5], the distribution of the owned region of each node is tightly approximated by an exponential distribution with mean $2^m / N$, where m is the number of bits of the Chord identifier space and N is the number of nodes in the network. With adaptive replication using replication factor k, each P-RLI node not only stores the mappings belonging to its owned region, but also replicates the mappings belonging to its k predecessors. Therefore, the number of mappings stored on each P-RLI node is determined by the sum of $k+1$ continuous owned regions before the node. Since the node identifiers are generated randomly, there is no dependency among those continuous owned regions. Intuitively, when the replication factor k increases, the sum of $k+1$ continuous owned regions will be more normally distributed. Therefore, we can achieve a better balance of mappings per node when we replicate more copies of each mapping. This hypothesis is verified by the simulation

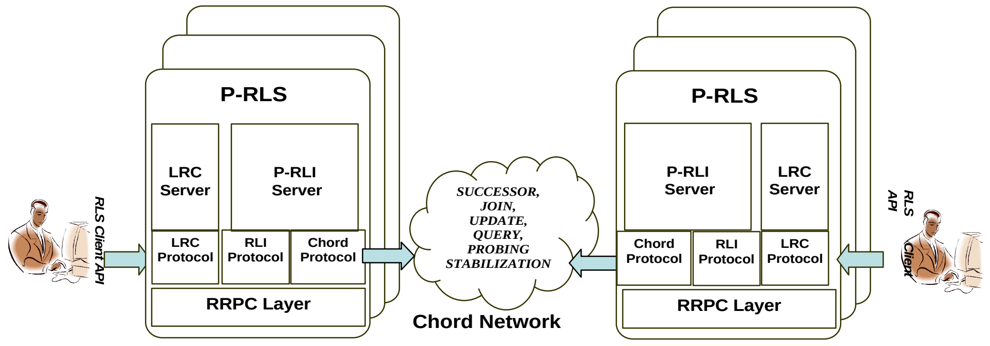

Figure 3: The P2P Replica Location Service Architecture

results in Section 0. Moreover, we can still use virtual nodes to distribute mappings among heterogeneous nodes with different capacities.

3.2.2 Query Load Balancing

Although successor replication can achieve better distribution of the mappings stored on P-RLI nodes, it does not solve the hotspot problem for extremely popular mappings. Consider a mapping {*"popular-object", rlsn://pioneer.isi.edu:8000*} that is queried 10,000 times from different P-RLI nodes. All the queries will be routed to the root node of the mapping, say node N_i, and it will be a query hotspot in the P-RLS network. The successor replication scheme does not solve the problem because all replicas of the mapping are placed on successor nodes that are *after* the root node (clockwise) in the circular space. The virtual nodes scheme does not solve this problem either because the physical node that hosts the virtual root node will be a hotspot.

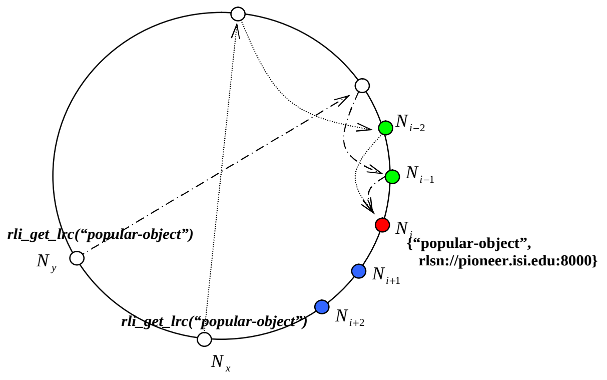
N_{i-2}
N_{i-1}
N_i
{"popular-object", rlsn://pioneer.isi.edu:8000}
N_{i+1}
N_{i+2}
rli_get_lrc("popular-object")
N_y
rli_get_lrc("popular-object")
N_x

Figure 4: P-RLI Queries for logical name "popular-object" traverse the predecessors of the root node N_i.

However, recall that in the Chord successor routing algorithm, each hop from one node to the next node covers at least half of the identifier space (clockwise) between that node and the destination successor node, i.e. the root node of the mapping. When the query is closer to the root node, there are fewer nodes in the circular space being skipped for each hop. Therefore, before the query is routed to its root node, it will traverse one of the predecessors of the root node with very high probability, as shown in Figure 4.

Therefore, we can improve our adaptive replication scheme and balance the query load for popular mappings by replicating mappings in the predecessor nodes of the root node. When a predecessor node of the root node receives a query to that root node, it will resolve it locally by looking up the replicated mappings and then return the query results directly without forwarding the query to the root node. We call this approach *predecessor replication.*

The predecessor replication scheme does not introduce extra overhead for Chord membership maintenance because each P-RLI node has information about its predecessors, since it receives probe messages from its predecessors. Also, this scheme has the same effect of evenly distributing mappings as the successor replication scheme because now each node stores its own mappings and those of its *k* successors.

4. THE P-RLS IMPLEMENTATION

We implemented a prototype of the P-RLS system by extending the RLS implementation in Globus Toolkit 3.0 with Chord protocols. Figure 3 shows the architecture of our P-RLS implementation. In this implementation, each P-RLS node consists of a LRC server and a P-RLI server. The LRC server implements the same LRC protocol in original RLS, but uses Chord protocol to update {*logical name, LRC*} mappings. The P-RLI server implements both original RLI protocol and the Chord protocol. Messages in the Chord protocol include *SUCCESSOR*, *JOIN*, *UPDATE*, *QUERY*, *PROBING*, and *STABILIZATION* messages. The *SUCCESSOR* message is routed to the successor node of the key in the message, and the node identifier and address of the successor node are returned to the message originator. When a P-RLI node joins the P-RLS network, it first finds its immediate successor node by sending a *SUCCESSOR* message, and then it sends the *JOIN* message directly to the successor node to join the network. The *UPDATE* message is used to add or delete a mapping, and the *QUERY* message is used to lookup matched mappings for a logical name. The P-RLI nodes also periodically send *PROBING* and *STABILIZATION* messages to detect node failures and repair the network topology.

We implemented the Chord successor lookup algorithm using the recursive mode rather than the iterative mode. In iterative mode, when a node receives a successor request for an object key, it will send information about the next hop to the request originator if it is not the successor node of the key. The originator then sends the request to the next node directly. By contrast, in recursive mode, after a node finds the next hop, it will forward the request to that node on behalf of the request originator. There are two approaches for the successor node to send the reply to the request originator. The first approach is simply to send the reply to the request originator. This approach might introduce a large number of TCP connections on the request originator from many different repliers. The second approach is to send the reply to its upstream node (the node where this node receives the successor request) and let the upstream node route the reply back to the request originator. In our P-RLS implementation, we implemented the second approach to avoid too many open TCP connections. All LRC, RLI and Chord protocols are implemented on top of the RLS RPC layer called RRPC.

5. P-RLS PERFORMANCE

In this section, we present performance measurements for a P-RLS system deployed in a 16-node cluster as well as analytical and simulation results for a P-RLS system ranging in size from 10 to 10,000 nodes with 500,000 {*logical name, LRC*} mappings.

5.1 Scalability Measurements

First, we present performance measurements for update operations (add or delete) and query operations in a P-RLS network running on our 16-node cluster. The cluster nodes are dual Pentium III 547MHz processors with 1.5 gigabytes of memory running Redhat Linux 9 and connected via a 1-Gigabit Ethernet switch. Figure 5 shows that update latency increases *O*(log *N*) with respect to the network size *N*. This result is expected, since in the Chord overlay network, each update message will be routed through at most *O*(log *N*) nodes. The error bar in the graph shows the standard deviation of the update latency. These results are measured for a P-RLS network that contains no mappings at the beginning of the test. Our test performs 1000 updates on each node, and the mean update latency and standard deviation are calculated. The maximum number of mappings in the P-RLS network during this test is 1000, with subsequent updates overwriting earlier ones for the same logical names.

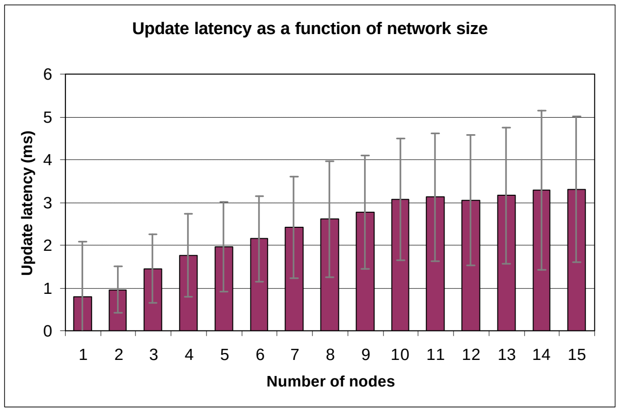

Figure 5: Shows update latency in milliseconds for performing an update operation in the P-RLS network.

Figure 6 shows that the query latency also increases on a log scale with the number of nodes in the system. These results are measured for two P-RLS networks that preload 100,000 and 1 million mappings respectively at the beginning of the test. Our test performs 1000 queries on each node, and the mean query latency and standard deviation are calculated. The results show that there is only slight latency increase when we increase the number of mappings in the P-RLS network from 100,000 to 1 million. This is because all the mappings on each node are stored in a hash table and the local lookup cost is nearly constant with respect to the number of mappings.

Next, Figure 7 shows the number of RPC calls that are required to perform a fixed number of updates as the size of the network increases. This test uses 15 clients with 10 requesting threads per client, where each thread performs 1000 update operations. For each configuration, the clients are distributed among the available P-RLI nodes. For a P-RLS network consisting of a single node, the number of RPC calls is zero. The number of RPC calls increases on a log scale with the number of nodes in the system. Since the queries are routed to the root node, the number of RPC calls for each query should increase logarithmically with the number of nodes in the system.

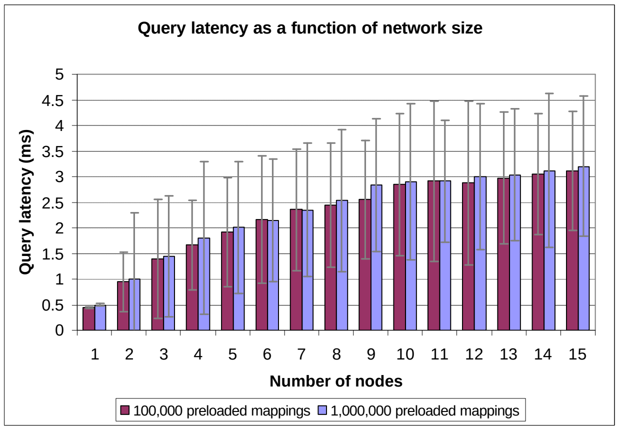

Figure 6: Shows query latency in milliseconds for performing a query operation in the P-RLI network

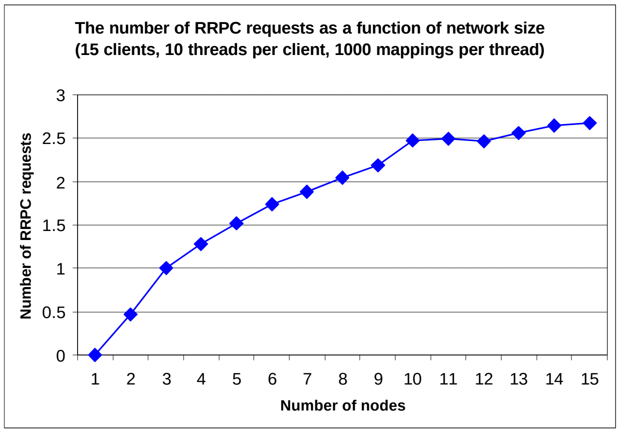

Figure 7: Number of RPC Calls performed for a fixed number of update operations as the size of the P-RLS network increases.

In the Chord overlay network, each P-RLI node must maintain pointers to its successors and to its finger nodes. The number of successors maintained by each node is determined by the replication factor *k* that is part of the P-RLI configuration. Figure 8 shows the rate at which the number of pointers to neighbors maintained by a P-RLI node increases. In this experiment, we set the replication factor to be two, i.e. each P-RLI node maintains the pointers to two successors. The number of neighbor pointers maintained by a node increases logarithmically with the size of the network. The error bars shows the minimum and maximum number of neighbor pointers maintained by each P-RLI node.

Next, we show the amount of overhead required to maintain the Chord overlay network. To maintain the Chord ring topology, P-RLS nodes periodically send probe messages to one another to determine that nodes are still active in the network. P-RLI nodes also send Chord stabilization messages to their immediate successors; these messages ask nodes to identify their predecessor nodes. If the node's predecessor has changed because of the addition of new P-RLI nodes, this allows the ring network to adjust to those membership changes. Finally, additional messages are sent periodically to maintain an updated finger table, in which each P-RLI node maintains pointers to nodes that are logarithmically distributed around the Chord identifier space. We refer collectively to these three types of messages for P-RLI membership maintenance as *stabilization traffic*.

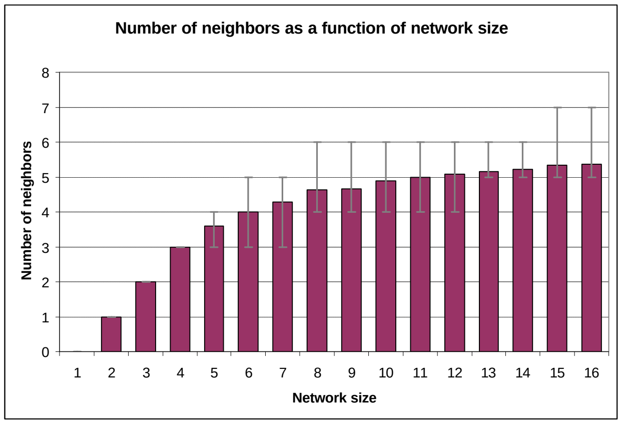

Figure 8: Rate of increase in pointers to neighbor nodes maintained by each P-RLI node as the network size increases, where the replication factor *k* equals to two.

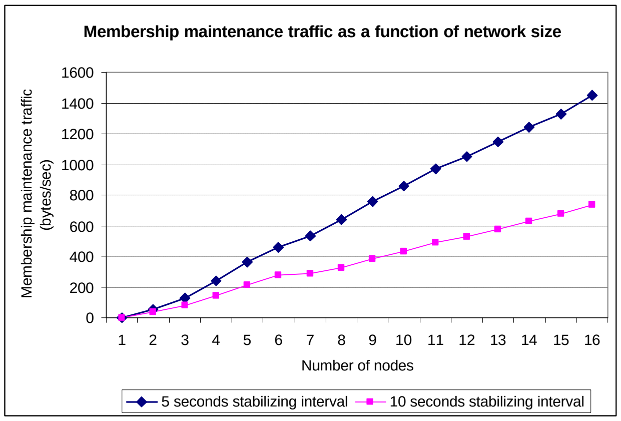

Figure 9: Stabilization Traffic for a P-RLS network of up to 16 nodes with stabilization intervals of 5 and 10 seconds.

Figure 9 shows the measured overhead in bytes per second for stabilization traffic as the number of nodes in the P-RLS network increases. The two lines show different periods (5 seconds and 10 seconds) at which the stabilization messages are sent. For both update intervals, the stabilization traffic is quite low (less than 1.5 Kbytes/second for 16 nodes). The stabilization traffic increases at a rate of $O(N \log N)$ for a network of size N. The graph shows the tradeoff between frequent updates of the Chord ring topology and stabilization traffic. If the stabilization operations occur more frequently, then the Chord overlay network will react more quickly to node additions or failures. In turn, this will result in better performance, since the finger tables for routing will be more accurate. The disadvantage of more frequent stabilization operations is the increase in network traffic for these messages.

5.2 Analytical Model for Stabilization Traffic

Next, we developed an analytical model for stabilization traffic in a P-RLS network to estimate the traffic for larger networks than we could measure directly.

Suppose we have a P-RLS network of N nodes with stabilization interval I and replication factor k. The average sizes of messages sent by a node to probe its neighbors, stabilize its immediate successor, and update its fingers are $\overline{S_p}$, $\overline{S_s}$, and $\overline{S_f}$ respectively. In our implementation, over the course of three stabilization intervals, each P-RLS node sends messages of these three types. Thus, the total membership maintenance traffic T for a stable network is:

$$T = \frac{(\log(N)+k)\times\overline{S_p} + \log(N)\times\overline{S_f} + \overline{S_s}}{3I} N$$

We measured the average message sizes in our P-RLS implementation. These values are shown in Table 1.

Table 1: Measured message sizes for our P-RLS implementation

$\overline{S_p}$	96.00
$\overline{S_f}$	164.73
$\overline{S_s}$	255.78

Based on this analytical model, we computed the membership traffic for networks ranging from 10 to 10000 nodes, where the replication factor is 2. These values are shown in Table 2. To validate our analytical model, we compared the calculated stabilization traffic with the traffic we measured in our 16-node cluster (shown in Figure 9 of the previous section). Figure 10 shows that the analytical model does a good job in predicting the stabilization traffic for a network of up to 16 P-RLI nodes.

Table 2: Stabilization traffic (bytes per second) predicted by our analytical model

Network size	Stabilization Interval	
	5 seconds	10 seconds
10	876 bytes/sec	438 bytes/sec
100	14533	7267
1000	203077	101538
10,000	2608190	1304095

5.3 Simulations for Adaptive Replication

In this section, we present simulation results for a larger network of P-RLI nodes. We simulate P-RLS networks ranging in size from 10 to 10,000 nodes with 500,000 mappings in the system. We picked 500,000 unique mappings as a representative number for a medium size RLS system. RLS deployments to date have ranged from a few thousand to tens of millions of mappings.

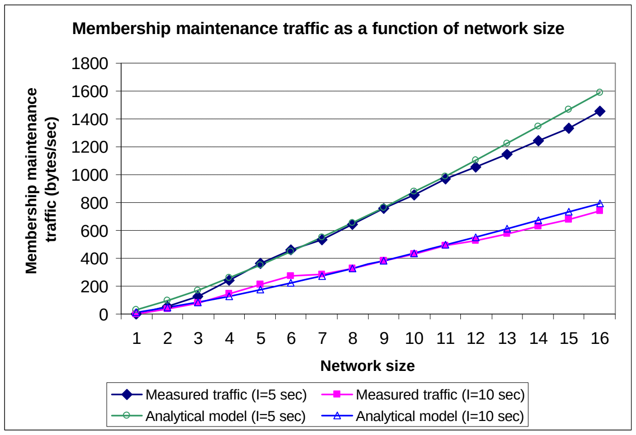

Figure 10: Comparison of measured and predicted values for stabilization traffic.

We used different random seeds and ran the simulation 20 times. In these simulations, we are interested in evaluating the distribution of mappings in a P-RLS network. A fairly even distribution of mappings in the P-RLS network should result in better load-balancing of queries to the system if those queries are uniformly distributed on the mappings. We also evaluate the distribution of queries for extremely popular mappings when we replicate mappings on the predecessors.

The simulator used in this section is written in Java. It is not a complete simulation of the P-RLS system, but rather, it focuses on how keys are mapped to the P-RLI nodes and how queries for mappings are resolved in the network.

First, we simulate the effect of increasing the number of replicas for each mapping in the P-RLS network, where *k* is the replication factor and there are a total of *k+1* replicas of each mapping. As we increase the replication factor, we must obviously store a proportionally increasing number of mappings in the P-RLS network. Table 3 shows the mean number of mappings per node for P-RLS networks ranging in size from 10 to 10,000 nodes when the replication factor *k* ranges from 0 to 12. We simulate a P-RLS network with a total of 500,000 unique mappings. The table shows that as the P-RLS network size increases, the average number of mappings per node decreases proportionally. As the replication factor increases, the average number of mappings per node increases proportionally. The mean numbers of mappings shown in Table 3 would increase proportionally with the total number of unique mappings in the P-RLS system.

Table 3: Mean number of mappings per node for a given network size and replication factor.

Network size	Replication Factor (Total Replicas)			
	0 (1)	1 (2)	4 (5)	12 (13)
10	50000	100000	250000	N/A
100	5000	10000	25000	65000
1000	500	1000	2500	6500
10000	50	100	250	650

While Table 3 shows that the mean number of mappings per node is proportional to the replication factor and inversely proportional to the network size, the actual distribution of mappings among the nodes is not uniform. However, we observe in Figure 11 that as the replication factor increases, the mappings tends to become more evenly distributed. Figure 11 shows the distribution of mappings over a network of 100 P-RLI nodes. On the horizontal axis, we show the number of nodes ordered from the largest to the smallest number of mappings per node. On the vertical axis, we show the cumulative percentage of the total mappings that are stored on some percentage of the P-RLI nodes. For a replication factor of zero (i.e., a single copy of each mapping), the 20% of the nodes with the most mappings contain approximately 50% of the total mappings. By contrast, with a replication factor of 12 (or 13 total replicas), the 20% of nodes with the most mappings contain only about 30% of the total mappings. Similarly, in the case of a single replica, the 50% of nodes with the most mappings contain approximately 85% of the total mappings, while for 13 total replicas, 50% of the nodes contain only about 60% of the total mappings.

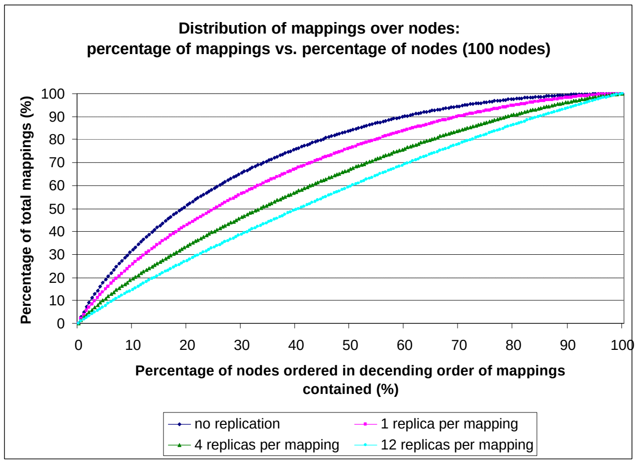

Figure 11: Shows the distribution of mappings among nodes in a P-RLS network. The vertical axis shows cumulative percentage of total mappings stored. The horizontal axis shows the percentage of total nodes, where the nodes are ordered from the most to least number of mappings per node. The replication factor varies from 0 to 12. There are 100 nodes in the P-RLS network and 500,000 unique mappings.

Figure 12 also provides evidence that as we increase the number of replicas for each mapping, the mappings are more evenly distributed among the P-RLI nodes. The vertical axis shows the cumulative density functions for the number of mappings stored per P-RLI node versus the number of mappings per node. The replication factor for P-RLI mappings ranges from 0 to 12, and the P-RLS network size is 100 nodes. The left-most line shows the case where P-RLI mappings are not replicated at all. This line shows a skewed distribution, in which most nodes store few mappings but a small percentage of nodes store thousands of mappings. By contrast, the line representing a replication factor of 12 is less skewed and resembles a Normal distribution. The ratio between the nodes with the least and greatest number mappings is approximately 3, with most nodes containing 40,000 to 100,000 mappings.

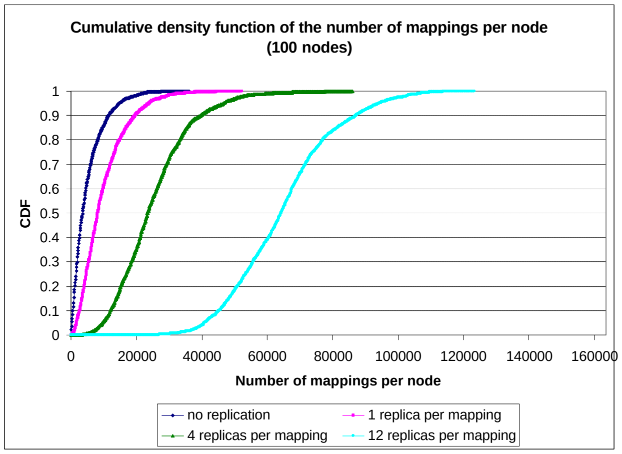

Figure 12: Cumulative distribution of mappings per node as the replication factor increases in a P-RLS network of 100 nodes with 500,000 unique mappings.

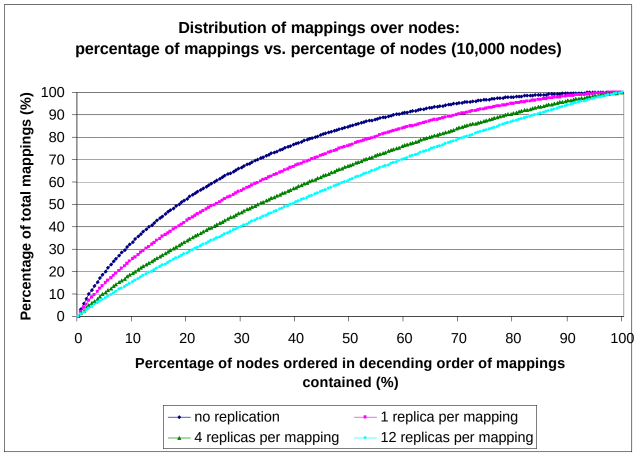

Figure 13: Shows the distribution of mappings among nodes in a P-RLS network. There are 10,000 nodes in the P-RLS network and 500,000 unique mappings.

Figure 11 and Figure 12 show a P-RLS network of 100 nodes. We ran the same simulations for network sizes of 1,000 and 10,000 nodes and found very similar results. For example, Figure 13 and Figure 14 show that a network of 10,000 P-RLS nodes has almost exactly the same shape as the previous graphs. The main difference between Figure 12 and Figure 14 is that the values on the horizontal axis showing the number of mappings per node differ by a factor of 100, corresponding to the difference in total nodes between the two P-RLS networks.

Next, we show in Figure 15 that the ratio between the greatest number of mappings per node and the average number per node converges as we increase the replication factor. The top line in Figure 15 shows this ratio, which decreases from approximately 10 to 2 as we increase the total number of replicas per mapping from 1 to 13. The graph shows the average ratio of 20 simulation runs. The bottom line in the graph shows that the ratio between the P-RLI node with the smallest number of mappings and the average number of mappings increases very slightly from a value of 0 as the replication factor increases.

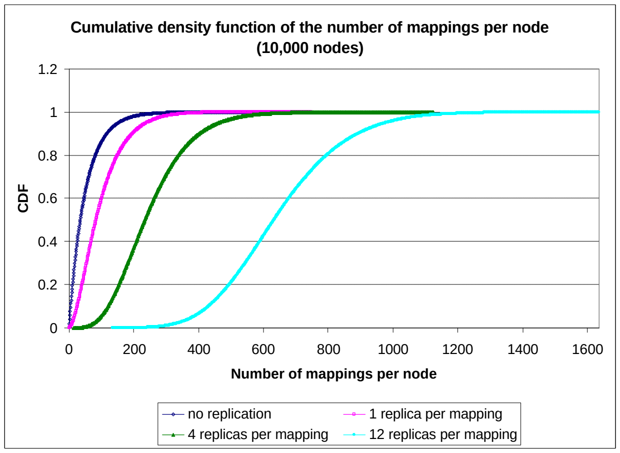

Figure 14: Cumulative density function for the number of mappings per node for a P-RLI network of 10,000 nodes for a given replication factor.

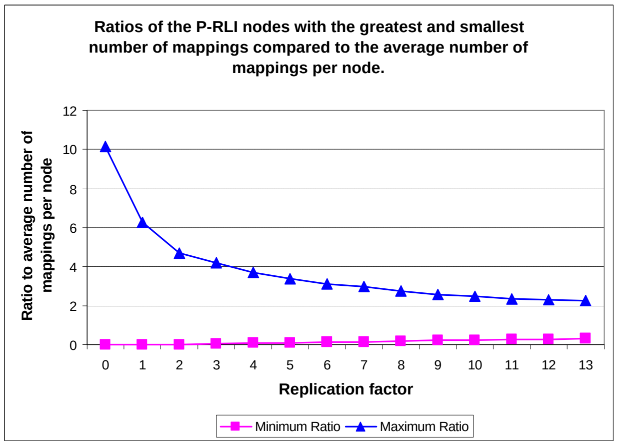

Figure 15: Ratios of the P-RLI nodes with the greatest and smallest number of mappings compared to the average number of mappings per node.

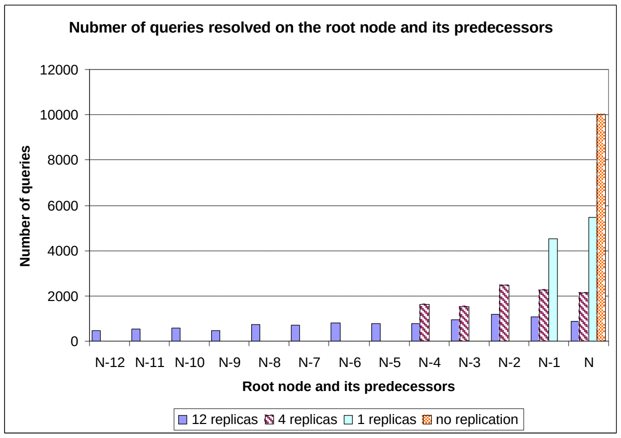

Figure 16: The number of queries resolved on the root node *N* and its predecessors becomes more evenly distributed when the number of replicas per mapping increases.

Finally, we present simulation results for our predecessor replication scheme, which is designed to reduce query hotspots for popular mappings. Figure 16 shows simulation results for a P-RLS network with 10,000 nodes. We randomly choose 100 popular mappings. For each of these popular mappings, we issue 10,000 queries from a randomly selected P-RLI node, and the average number of queries that are resolved on the root node and its predecessors is simulated. In Figure 16, node *N* represents the root node and node *N-i* is the *i*-th predecessor of node *N* in the Chord overlay network. Thus, node *N-1* is the immediate predecessor of *N*, and node *N-12* is 12th predecessor of node *N*. The results show that if there is no predecessor replication, all 10,000 queries will be resolved by the root node *N*. However, as we increase the number of replicas of popular mappings on node *N*'s predecessors, the query load is more evenly distributed among node *N* and its predecessors.

6. RELATED WORK

6.1 Replica Management in Grids

Related Grid systems include the Storage Resource Broker [19] and GridFarm [20] projects that register and discover replicas using a metadata service and the European DataGrid Project [21], which has implemented a Web Service-based Replica Location Service that is based on the RLS Framework [1]. Ripeanu et. al [22] constructed a peer-to-peer overlay network of replica location services; their work focused on the use of Bloom filter compression for soft state updates for efficient distribution of location information. In contrast to P-RLS, which inserts mappings and routes queries in a structured peer-to-peer network, Ripeanu's approach distributes compressed location information to each node participating in an unstructured peer-to-peer network. Each node can answer queries locally without forwarding requests, and thus query latencies are reduced compared to P-RLS. However, since updates have to be propagated to the whole network, the cost of creating and deleting a replica mapping in Ripeanu's unstructured scheme is higher than for P-RLS when the network scales to large sizes.

6.2 Replica Management in Distributed File Systems and Distributed Databases

Data replication has also been studied extensively in the literature of distributed file systems and distributed databases [23-28]. A primary focus of much of that work is the tradeoff between the consistency and availability of replicated data when the network is partitioned. In distributed file systems, the specific problem of replica location is known as the replicated volume location problem, i.e. locating a replica of a volume in the name hierarchy [29]. NFS [30] solves this problem using informal coordination and out-of-band communication among system administrators, who manually set mount points referring to remote volume locations. Locus [31] identifies volume locations by replicating a global mounting table among all sites. ASF [32] and Coda [33] employ a Volume Location Data Base (VLDB) for each local site and replicate it on the backbone servers of all sites. Ficus [29] places the location information in the mounted-on leaf, called a *graft point*. The graph points need location information since they must locate a volume replica in the distributed file system. The graft points may be replicated at any site where the referring volume is also replicated.

6.3 Structured Peer-to-Peer Networks

Besides Chord, there are many other structured Peer-to-Peer networks proposed in recent years, such as Tapestry [15], Pastry [16], CAN [17], Koorde [18], Skip Graphs [34] and SkipNet [35].

The routing algorithms used in Tapestry and Pastry are both inspired by Plaxton [36] . The idea of the Plaxton algorithm is to find a neighboring node that shares the longest prefix with the key in the lookup message and to repeat this operation until a destination node is found that shares the longest possible prefix with the key. Each node has neighboring nodes that match each prefix of its own identifier but differ in the next digit. For a system with *N* nodes, each node has $O(\log N)$ neighbors, and the routing path takes at most $O(\log N)$ hops. Tapestry uses a variant of the Plaxton algorithm and focuses on supporting a more dynamic environment, with nodes joining and leaving the system. It maintains neighborhood state through both proactive, explicit updates and soft-state republishing. To adapt to environment changes, Tapestry dynamically selects neighbors based on the latency between the local node and its neighbors. Pastry uses a prefix-based lookup algorithm similar to Tapestry's. Each Pastry node maintains a routing table, a neighborhood set and a leaf set. Pastry also employs the locality information in its neighborhood set to achieve topology-aware routing, i.e. to route messages to the nearest node among the numerically closest nodes [37].

CAN [17] maps its keys to a *d*-dimensional Cartesian coordinate space. The coordinate space is partitioned into *N* zones for a network with *N* nodes. Each CAN node owns the zone corresponding to the mapping of its node identifier in the coordinate space. The neighbors on each node are the nodes that own the contiguous zones to its local zone. Routing in CAN is straightforward: a message is always greedily forwarded to a neighbor that is closer to the key's destination in the coordinate space. Each node in a CAN network with *N* nodes has $O(d)$ neighbors, and routing path length is $O(dN^{1/d})$ hops. Compared to Tapstry/Pastry and Chord, CAN keeps less neighborhood state when *d* is less than $O(\log N)$. However, CAN has relatively longer routing paths on lookup operations in this case. If *d* is chosen to be $O(\log N)$, it has $O(\log N)$ neighbors and $O(\log N)$ routing hops like the above algorithms. CAN trades off neighborhood state for routing efficiency by adjusting the number of dimensions.

The above DHT algorithms are quite scalable because of their logarithmic neighborhood state and routing hops. However, these bounds are close to optimal but not optimal. Kaashoek et al. proved that for any constant neighborhood state *k*, $\Theta(\log N)$ routing hops is optimal. But in order to provide a high degree of fault tolerance, a node must maintain $O(\log N)$ neighbors. In that case, $O(\log N / \log\log N)$ optimal routing hops can be achieved. Koorde is a neighborhood state optimal DHT based on Chord and de Bruijn graphs. It embeds a de Bruijn graph on the identifier circle of Chord for forwarding lookup requests. Each node maintains two neighbors: its successor and the first node that precedes its first de Bruijn node. It meets the lower bounds, such as $O(\log N)$ routing hops per lookup request with only 2 neighbors per node. To allow users to trade-off neighbor state for routing hops, Koorde can use degree-k de Bruijn graphs. When $k = \log N$, Koorde can be made fault-tolerant, and the number of routing hops is $O(\log N / \log\log N)$.

Recently, two novel, structured P2P systems based on skip lists [38] were proposed: Skip Graphs [34] and SkipNet [35]. These systems are designed for use in searching P2P networks and provide the ability to perform queries based on key ordering, rather than just looking up a key. Thus, Skip Graphs and SkipNet maintain data locality, unlike DHTs. Each node in a Skip Graphs or SkipNet system maintains $O(\log N)$ neighbors in its routing table. A neighbor that is 2^h nodes away from a particular node is said to be at level *h* with respect to that node. This scheme is similar to the fingers in Chord. There are 2^h rings at level *h* with $n/2^h$ nodes per ring. A search for a key in Skip Graphs or SkipNet begins at the top-most level of the node seeking the key. It proceeds along the same level without overshooting the key, continuing at a lower level if required, until it reaches level 0. The number of routing hops required to search for a key is $O(\log N)$. In addition, these schemes are highly resilient, tolerating a large fraction of failed nodes without losing connectivity.

7. FUTURE WORK

We plan to conduct further performance experiments for the P-RLS system, including measuring the throughput of the system for update and query operations at high request loads and the effect of adaptive replication on query load balancing. We will also measure the fault tolerance of the system.

Our current results focus on P-RLS performance in a local area network, but we plan similar studies in the wide area. One important issue in the wide area is potentially long latencies for sending messages among nodes in the P-RLS network. One concern regarding the use of Chord in a wide area deployment is that each hop in the Chord overlay might correspond to multiple hops in the underlying IP network. We plan to experiment with the algorithm proposed by Zhang et al [39] called lookup-parasitic random sampling (LPRS) that reduces the IP layer lookup latency of Chord. The authors proved that LPRS-Chord can result in lookup latencies proportional to the average unicast latency of the network, provided the underlying physical topology has power-law latency expansion. We plan to use their algorithm to reduce the network latencies experienced by a wide area P-RLS system.

Providing security in an open peer-to-peer network is an open problem [40]. Castro et al. [41] combine secure node identifier assignment, secure routing table maintenance, and secure message forwarding to tolerate up to 25% malicious nodes in a peer-to-peer network. However, mechanisms for providing access control for P-RLS mappings are still needed. We will be addressing this problem in our future work.

8. CONCLUSIONS

We have described the design of a Peer-to-Peer Replica Location Service. The goal of our design is to provide a distributed RLI index with properties of self-organization, fault-tolerance and improved scalability. The P-RLS design uses the overlay network of the Chord peer-to-peer system to self-organize P-RLS servers.

We described the P-RLS implementation and presented performance measurements and simulation results. Our performance measurements on a 16-node cluster demonstrated that update and query latencies increase at a logarithmic rate with the size of the P-RLS network. We also demonstrated that the overhead of maintaining the P-RLS network is reasonable, with the number of remote procedure calls and the number of pointers maintained by each P-RLS node increasing at a logarithmic rate with respect to the size of the network and the amount of traffic to stabilize the network topology increasing at a rate of $O(N \log(N))$.

We also presented simulation results for adaptive replication of P-RLS mappings for network sizes ranging from 10 to 10,000 P-RLS nodes. We demonstrated that as the replication factor of these mappings increases, the mappings are more evenly distributed among the P-RLI nodes when using the adaptive replication scheme. Also, we showed that the predecessor replication scheme can more evenly distribute the queries for extremely popular mappings, thereby reducing the hotspot effect on a root node.

9. ACKNOWLEDGEMENTS

We are grateful to Mats Rynge for his technical support of our work on the computing cluster; to Naveen Palavalli and Shishir Bharathi for helpful discussions on the RLS implementation; and to Matei Ripeanu for his valuable feedback on this paper. This work was supported in part by DOE Cooperative Agreements DE-FC02-01ER25449 (SciDAC- DATA) and AFOSR funding under grant F49620-01-1-0341.

10. REFERENCES

[1] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunst, M. Ripeanu, B, Schwartzkopf, H, Stockinger, K. Stockinger, B. Tierney, "Giggle: A Framework for Constructing Sclable Replica Location Services," presented at SC2002 Conference, Baltimore, MD, November 2002.

[2] A. L. Chervenak, Naveen Palavalli, Shishir Bharathi, Carl Kesselman, Robert Schwartzkopf, "Performance and Scalability of a Replica Location Service," presented at High Performance Distributed Computing Conference (HPDC-13), Honolulu, HI, June 2004.

[3] "The Earth Systems Grid." http://www.earthsystemsgrid.org.

[4] "LIGO - Laser Interferometer Gravitational Wave Observatory." http://www.ligo.caltech.edu/.

[5] I. Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," presented at ACM SIGCOMM, 2001.

[6] B. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of ACM*, vol. 13, pp. 422-426.

[7] P. Avery and I. Foster, "The GriPhyN Project: Towards Petascale Virtual Data Grids," 2001. www.griphyn.org.

[8] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, "Pegasus: Planning for Execution in Grids," GriPhyN Project Technical Report 2002-20.

[9] E. Deelman, et. al, "Mapping Abstract Complex Workflows onto Grid Environments," *Journal of Grid Computing*, vol. 1, pp. 25-39.

[10] "Gnutella." http://gnutella.wego.com.

[11] M. Ripeanu, I. Foster, and A. Iamnitchi., "Mapping the Gnutella network: properties of large-scale peer-to-peer systems and implications for system design," *IEEE Internet Computing Journal*.

[12] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems," presented at Multimedia Computing and Networking, 2002.

[13] S. Sen, Jia Wong, "Analyzing peer-to-peer traffic across large networks," presented at Proceedings of the Second ACM SIGCOMM Workshop on Internet Measurment,, November 2002.

[14] S. Ratnasamy, S. Shenker, and I. Stoica, "Routing algorithms for DHTs: Some open questions," presented at IPTPS02, Cambridge, USA, March 2002.

[15] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-resilient wide-area location and routing," U.C. Berkeley, Berkeley Technical Report UCB-CSD-01-1141, April 2001.

[16] A. Rowstron, P. Druschel., "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," presented at International Conference on Distributed Systems Platforms (Middleware), November 2001.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," presented at ACM SIGCOMM, August 2001.

[18] F. Kaashoek, David R. Karger, "Koorde: A Simple Degree-optimal Hash Table," presented at 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), February, 2003.

[19] C. Baru, R. Moore, et al., "The SDSC Storage Resource Broker," presented at CASCON'98 Conference.

[20] O. Tatebe, et al., "Worldwide Fast File Replication on Grid Datafarm," presented at 2003 Computing in High Energy and Nuclear Physics (CHEP03), March 2003.

[21] L. Guy, P. Kunszt, E. Laure, H. Stockinger, K. Stockinger, "Replica Management in Data Grids," presented at Global Grid Forum 5.

[22] M. Ripeanu, Ian Foster, "A Decentralized, Adaptive, Replica Location Mechanism," presented at 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), Edinburgh, Scotland, June, 2002.

[23] Y. Breitbart, H. Korth, "Replication and Consistency: Being Lazy Helps Sometimes," presented at 16th ACM SIGACT/SIGMOD Symposium on the Principles of Database Systems, Tucson, AZ, 1997.

[24] J. Gray, P. Helland, P. O'Neil, D. Shasha, "The Dangers of Replication and a Solution," presented at ACM SIGMOD Conference, 1996.

[25] K. Petersen, et al., "Flexible Update Propagation for Weakly Consistent Replication," presented at 16th ACM Symposium on Operating Systems Principles (SOSP-16), Saint Malo, France, 1997.

[26] D. B. Terry, K. Petersen, M. J. Spreitzer, and M. M. Theimer, "The Case for Non-transparent Replication: Examples from Bayou," presented at 14th International Conference on Data Engineering, 1998.

[27] Wiesmann, M., F. Pedone, A. Schiper, B. Kemme, G. Alonso, "Database Replication Techniques: A Three Paramater Classification," presented at 19th IEEE Symposium on Reliable Distributed Systems, Nuernberg, Germany, 2000.

[28] J. Sidell, P.M. Aoki, A. Sah, C. Staelin, M. Stonebraker, A. Yu, "Data Replication in Mariposa, "Data Replication in Mariposa," presented at 12th International Conference on Data Engineering, New Orleans, LA, 1996.

[29] J. T. W. Page, R. G. Guy, G. J. Popek, J. S. Heidemann, W. Mak, and D. Rothmeier, "Management of Replicated Volume Location Data in the Ficus Replicated File System," presented at USENIX Conference, 1991.

[30] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, "Design and implementation of the Sun Network File System," presented at USENIX Conference, June 1985.

[31] G. Popek, The Locus Distributed System Architecture: The MIT Press, 1986.

[32] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, D. C. Steere, "Coda: A Highly Available System for a Distributed Workstation Environment," IEEE Transactions on Computers, vol. 39(4), pp. 447-459, April 1990.

[33] E. R. Zayas, C. F. Everhart, "Design and Specification of the Cellular Andrew Environment," Carnegie-Mellon University Technical Report CMU-ITC-070, August 1988.

[34] J. Aspnes, Gauri Shah, "Skip Graphs," presented at Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 2003.

[35] N. Harvey, M. Jones, S. Saroiu, M.Theimer, and A. Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Properties," presented at Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03), Seattle, WA, March 2003.

[36] C. Plaxton, R. Rajaraman, A. Richa, "Accessing nearby copies of replicated objects in a distributed environment," presented at ACM SPAA, Newport, Rhode Island, June 1997.

[37] M. Castro, P. Druschel, Y. C. Hu, A. Rowstron, "Topology-aware routing in structured peer-to-peer overlay networks," presented at Intl. Workshop on Future Directions in Distributed Computing, June 2002.

[38] W. Pugh, "Skip Lists: A Probabilistic Alternative to BalancedTrees," presented at Workshop on Algorithms and Data Structures, 1989.

[39] H. Zhang, A. Goel, R. Govindan, "Incremental Optimization In Distributed Hash Table Systems," presented at ACM SIGMETRICS, 2003.

[40] E. Sit, R. Morris, "Security considerations for peer-to-peer distributed hash tables," presented at 1st International Workshop on Peer-to-Peer Systems (IPTPS '02), Cambridge, MA, March 2002.

[41] M. Castro, P. Drushel, A. Ganesh, A. Rowstron, and D. Wallach, "Secure routing for structured peer-to-peer overlay networks," presented at 5th Usenix Symposium on Operating Systems Design and Implementation (OSDI), Boston, MA, 2002.