Journal of Grid Computing (2006)
DOI: 10.1007/s10723-005-9009-1

Applying Peer-to-Peer Techniques to Grid Replica Location Services

Ann L. Chervenak[1,★] and Min Cai[2]
[1]*University of Southern California Information Sciences Institute, 4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292, USA*
E-mail: annc@isi.edu
[2]*University of Southern California, Computer Science Department, Los Angeles, CA 90089, USA*

Received 27 September 2005; accepted 30 November 2005

Key words: Grids, peer-to-peer systems, Replica Location Service, resource discovery service

Abstract

Peer-to-peer systems offer attractive system management properties, including the ability of components that join the network to self-organize; scalability up to tens of thousands of members; the ability of the network to automatically repair its topology after node failures; and techniques for maintaining redundant information to improve reliability and load balancing. We investigate applying peer-to-peer techniques to Grid services that are oriented toward resource discovery. In particular, we apply the Chord structured peer-to-peer overlay network to the Globus Replica Location Service, which allows registration and discovery of data replicas. We describe the design and performance of a Peer-to-Peer Replica Location Service (P-RLS) that uses the Chord algorithm to self-organize P-RLS servers and exploits the Chord overlay network to replicate P-RLS mappings adaptively. We present performance measurements and simulation results for the P-RLS system. We also discuss outstanding issues for applying peer-to-peer techniques to Grid resource discovery services.

Abbreviations: P2P – peer-to-peer; RLS – Replica Location Service; P-RLS – Peer-to-peer Replica Location Service; LRC – Local Replica Catalog; RLI – Replica Location Index

1. Introduction

As Grid deployments have increased in scale, system management has become increasingly challenging. System administrators must work to configure systems manually and to maintain and update them. Efforts such as the Grid Research Integration Deployment and Support (GRIDS) Center [4] seek to ease the burden of deploying and operating Grids by providing facilities for testing Grid software on a variety of operating system platforms, guaranteeing interoperability of software versions, integrating the installation process and providing support for easier configuration. Commercial efforts to provide more automatic system configuration and management include work on *autonomic* or *utility* computing [3, 31].

While these efforts are valuable, much of the configuration and maintenance of Grids is still done manually, primarily because individual Grid services generally do not have capabilities of self-configuration, reliability or self-healing after component failures or unexpected events. This situation makes Grids complex and expensive to deploy and maintain and impedes the adoption of Grid systems. Making individual Grid services and components more self-configuring, reliable and scalable would provide tremendous benefits to both scientific and commercial communities.

★ Corresponding author.

In this work, we investigate applying peer-to-peer (P2P) techniques to Grid services that are oriented toward resource discovery. Peer-to-peer systems offer attractive system management properties, including the ability of components that join the network to self-organize and discover other components; scalability of peer-to-peer networks up to tens of thousands of members; the ability of the network to automatically repair its topology after node failures or departures from the system; and techniques for maintaining redundant information in the peer-to-peer network to improve reliability and load balancing.

Grid resource discovery services are particularly well-suited to the application of peer-to-peer techniques. These discovery services share the requirement of maintaining a group of distributed indexes and requiring state updates among these indexes, and thus have similar requirements to the Internet file sharing and discovery application that is the basis of most peer-to-peer research. Grid resource discovery services differ from one another in the types of resource information they store and the consistency requirements and rates of change for that information. In this paper, we apply P2P techniques to Grid Replica Location Services [18, 19], which provide capabilities for registration and discovery of replicated data items in a Grid environment. Other resource discovery services that could benefit from peer-to-peer techniques include distributed Grid information services, such as the Globus Monitoring and Discovery Service [21, 59], and distributed metadata services.

In this paper, we describe the Peer-to-Peer Replica Location Service (P-RLS), which uses a structured peer-to-peer network to organize a distributed index of replica location mappings. The P-RLS design uses the structured overlay network of the Chord peer-to-peer system [53] to self-organize P-RLS servers. We implemented a prototype of the P-RLS system by extending the RLS implementation in Globus Toolkit Version 3.0 with Chord protocols. We measured 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.

The paper begins with a description of the Globus Replica Location Service. Next, we provide an overview of peer-to-peer systems and a description of the Chord structured overlay network. Next, we describe the P-RLS design and implementation and present performance measurements and simulation results. We discuss related work and conclude the paper with a discussion of outstanding research issues for applying peer-to-peer techniques to Grid systems. These include adapting peer-to-peer algorithms for the typical scale and dynamism of Grid resource discovery systems; addressing security issues such as authentication and authorization; evaluating different peer-to-peer schemes, including structured and unstructured networks; and investigating how well the queries supported by peer-to-peer systems match the requirements of Grid resource discovery services.

2. The Globus Replica Location Service

In Grid environments, data intensive applications often replicate data for reasons of fault tolerance as well as to improve performance by allowing access to multiple replicas. A Replica Location Service (RLS) is one component of a Grid data management system that provides functionality to register and discover data replicas. When a user creates a new replica of a data object, the user also registers the existence of the replica in the RLS by creating an association between a logical name for the data item and the physical location of the replica. An RLS client discovers data replicas by querying the catalog based on logical identifiers for data, its physical location or user-defined attributes associated with logical or physical names. In earlier work, Chervenak et al. [18] 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 available as part of the Globus Toolkit Versions 3 and 4. We have demonstrated that this RLS implementation provides good performance and scalability [19].

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.

The RLS framework [18] includes a soft state maintenance mechanism and optional compression of state updates. 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 implements an optional Bloom filter compression scheme [13]. In this scheme, each LRC sends a bit map that summarizes its contents to the RLIs. The bit map is constructed by performing a series of hash functions on the logical names that are registered in an LRC and setting the corresponding bits in the bit map.

Finally, the RLS framework envisions a membership management service that keeps track of LRCs and RLIs as they enter and leave the system, including which servers send and receive soft data updates from one another, 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 the Earth System Grid [10] the Laser Interferometer Gravitational Wave Observatory [33], the CMS and ATLAS high energy physics applications [1], and the Quantum Chromodynamics Grid [5]. Several other applications use

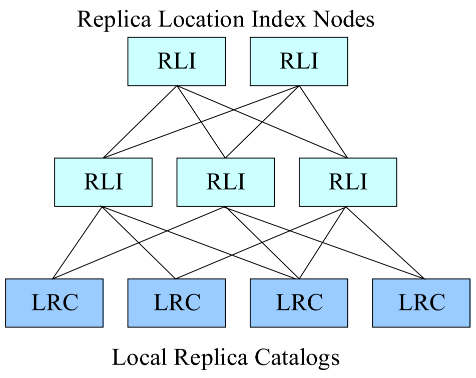

Figure 1. Example of a hierarchical RLI Index configuration supported by the RLS implementation available in the Globus Toolkit Version 3.

RLS *via* the Pegasus workflow management system [24, 25], including the National Virtual Observatory [11, 23] and the Southern California Earthquake Center [6].

Despite its successful deployment by these projects, there are several features of the existing RLS that could be improved. Because a membership service has not been implemented for the RLS, each deployment is statically configured, and the system does not automatically react to membership changes as servers join or leave the system. Configuration files at each RLS server specify parameters including authorization policies; whether a particular server acts as an LRC, an RLI or both; and how state updates are propagated from LRCs to RLIs. 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 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.

In addition, 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 not self-healing after RLI failures and is not able to maintain consistency guarantees for index contents. We have no ability to specify, for example, that we want the system to maintain at least three 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 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 C, Java and Python. The implementation supports two types of soft state updates from LRCs to RLIs: A complete list of logical names registered in the LRC and 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.

3. Peer-to-Peer Systems

We summarize recent work in peer-to-peer systems and then describe the Chord structured peer-to-peer overlay network.

Foster and Iamnitchi [26] define peer-to-peer systems as "decentralized, self-organizing distributed systems, in which all or most communication is symmetric." Peer-to-peer file sharing systems deployed on the Internet typically involve a large number of participants. For example, the Gnutella system is able to scale to hundreds of thousands of nodes [16, 43]. There are no centralized authorities to organize these networks. Instead, peer-to-peer systems self-organize, and each node is only connected to a small fraction of the other nodes or peers. In addition, the nodes in peer-to-peer systems tend to be highly dynamic, joining the network for relatively short periods. Different studies indicate that most nodes remain in the Gnutella network less than 24 h [44], for a mean time of 60 min [49], or that 50% of nodes are available 30% of the time over seven days [12].

For discovery of resources in a peer-to-peer network, systems have evolved from centralized indexing to flooding-based *unstructured* schemes and to Distributed Hash Tables (DHTs) and other *structured* approaches.

Systems like Napster maintain a centralized index server, and each resource discovery operation is performed by querying the central server. This approach has limited scalability and represents a single point of failure.

The flooding-based approach used in Gnutella [16, 43] constructs an *unstructured* overlay network among nodes. Queries are flooded to the whole network. This approach is decentralized and improves fault tolerance by eliminating the single point of failure. However, query flooding potentially introduces large message traffic and processing overheads in the network [45, 49, 51] and therefore may not scale well. To avoid flooding of the network, the number of hops on the forwarding path is typically bounded by the time-to-live (TTL) field of query messages. When using a TTL bound, search results are not deterministic, and this approach cannot guarantee finding desired files even if they exist. To make unstructured P2P systems more scalable, several random walk and replication schemes have been proposed to reduce the number of flooding messages, as in the Gia system [17, 34]. Most queries in an Internet file sharing application are for a small fraction of popular files; if those popular files are well-replicated, then they can easily be found with a random walk of queries without flooding queries to the entire network. However, it is still difficult for an unstructured peer-to-peer network to find files that are not popular and well-replicated. In practice, the system may bias discovery toward the most popular files or resources and may ignore rarely used resources.

To reduce the extra communication and processing overhead of flooding queries, distributed hash table (DHT) approaches were proposed to construct a structured overlay network among nodes and to utilize message routing instead of flooding to look up a resource [42]. DHT systems construct a variety of overlay networks and employ different routing algorithms. Depending on the overlay and routing, each node in these systems typically maintains $O(\log N)$ or d pointers to its neighbors in an N node network, where d is the dimension of the hypercube organization of the network. These systems can finish a lookup operation in $O(\log N)$ or $O(dN^{1/d})$ hops. Therefore, these DHT systems provide good scalability as well as failure resilience.

Besides efficient insertion and lookup, DHT-based systems have properties of self-organization for low maintenance cost and self-healing for failure resilience. For example, in the Chord [53] system, all nodes join the overlay network based on the positions of their node identifiers in the circular identifier space. Thus all nodes self-organize into a structured overlay network automatically. Whether a node is responsible for storing a key is based on the interval in the identifier space between its predecessor and itself. When a node leaves the network, its successor node will automatically detect the departure and take over the interval for which the departed node was responsible. This self-healing behavior is useful to provide redundancy and fault tolerance in large-scale peer-to-peer systems.

Distributed Hash Tables (DHTs) offer scalable lookup for distributed resources based on exact matches for a given key. However, they do not support several types of queries that are desirable in Grid resource discovery services, including multi-attribute queries, range queries, or richer queries using query languages such as X Path.

In structured peer-to-peer networks like Chord [53] and pure unstructured networks like Gnutella [2] and Freenet [20], all nodes have equal roles and responsibilities. Yang and Garcia-Molina [57] argue that searching in these networks tends to be inefficient. They propose a super-peer network that takes advantage of the heterogeneity of peers by assigning the role of super-peers to more powerful nodes, while less powerful nodes are clients of super-peers. The authors show that a super-peer network can improve performance by satisfying queries at the super-peer level. Mizrak et al. [35] show that this scheme can perform lookups in constant time, while the network scales to millions of nodes.

3.1. *The Chord Structured Peer-to-Peer Network*

Our design of the P-RLS uses the structured peer-to-peer overlay algorithms of the Chord system. Next, we describe the Chord design proposed by Stoica et al. [53]. 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 its 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 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 $1 \leq i \leq m$. The first finger node is the immediate successor of n, where $i = 1$. When node n wants to look up 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 maintains pointers to $O(\log N)$ neighbors.

The basic Chord algorithm does not necessarily result in an even distribution of mappings among 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. Thus, 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, so the number of mappings stored on each node may vary considerably. Chord achieves a more even distribution by associating object keys with *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 neighborhood information for 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 may be lost when a node fails. Section 4.1 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.

4. The P-RLS 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 3.0 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 peer-to-peer RLI (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 3.1. 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 P-RLS 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 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 those nodes. We call this successor node the *root node* of the mapping. Figure 2 shows how three mappings are placed in a P-RLS network with eight 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

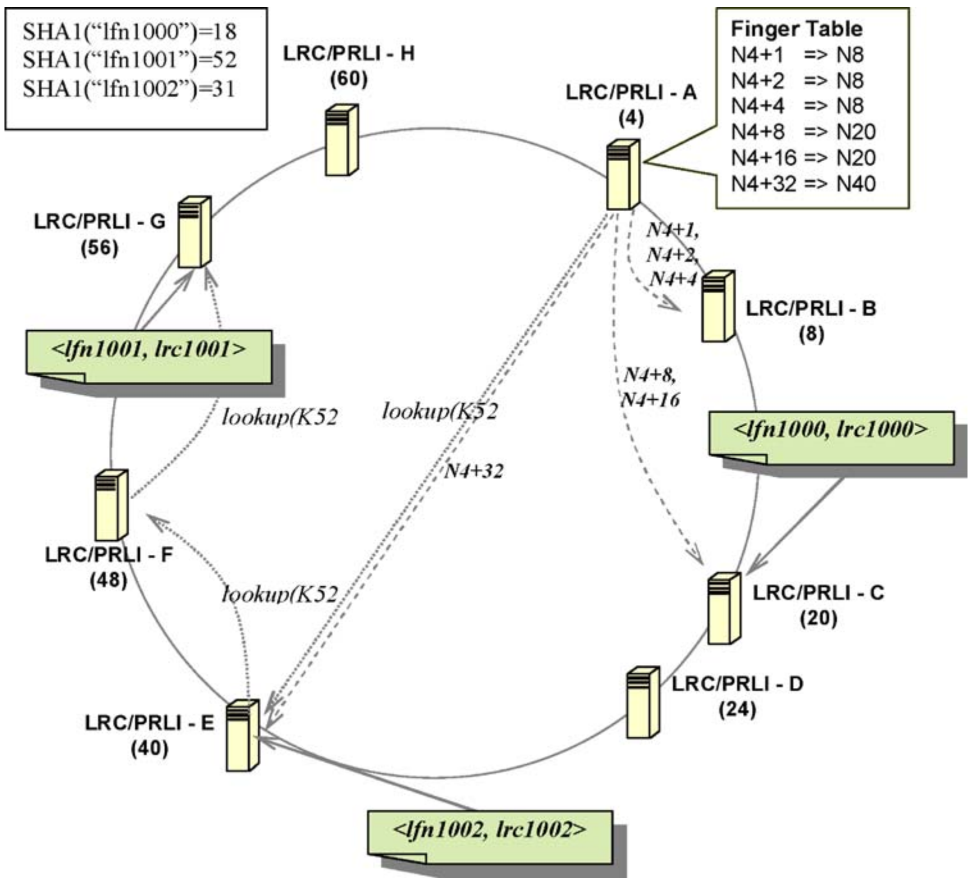

Figure 2. Example of the mapping placement of three mappings in the P-RLS network with eight nodes.

database and returns the query result to the client. Otherwise, the node will determine the successor node for the object key 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 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 additional aspects of our P-RLS design, including adaptive replication of P-RLI mappings on successor nodes, the effect of replication on the distribution of mappings, and replication of mappings on predecessor nodes to improve load balancing for popular queries.

4.1. *Adaptive Replication on Successor Nodes*

The P-RLI nodes in the P-RLS network can leave or fail at any time, or the network connection between any two nodes can be broken. To resolve queries for {*logical name, LRC*} mappings continuously despite node failures, we need to replicate mappings on multiple P-RLI nodes. The Chord membership maintenance protocol maintains the ring topology among P-RLI nodes even when a number of nodes join or leave concurrently. Thus, we replicate mappings in the P-RLS network based on the membership information provided by the Chord protocol.

In the P-RLS design, each mapping is stored on the root node of the mapping. The root node maintains connections to its successor nodes in the Chord ring for routing. A simple replication approach is to replicate the mappings stored on the root node to its k successors nodes, 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 $k + 1$.

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 departing 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 replicated mappings from that root node must be removed from the former successor node. We achieve this using the soft state replication and the periodic probing messages of the Chord protocol. Each mapping has an expiration time. When a node receives a probe message from its predecessor, it extends 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 removed. 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.

4.2. *Distribution of Mappings Among Nodes*

The Chord algorithm described in Section 3 can use a consistent hashing and virtual nodes to balance the number of keys stored on each node. However, the use of virtual nodes incurs some overhead, such as maintaining more neighbors per node and increasing the number of hops per lookup.

Our successor replication scheme, which adaptively replicates mappings on multiple P-RLS nodes for fault tolerance, can also improve the distribution of mappings among 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 [53], 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 results in Section 6.3. Moreover, we can still use virtual nodes to distribute mappings among heterogeneous nodes with different capacities.

4.3. *Query Load Balancing and Predecessor Replication*

While successor replication can achieve a more even distribution of the number of mappings stored on P-RLI nodes, it does not address the issue of query load balancing for popular mappings, which may generate a large number of queries for the root nodes of those 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 this 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.

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 3.

We can improve our adaptive replication scheme and balance the query load for popular mappings by replicating mappings on 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 already 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, be-

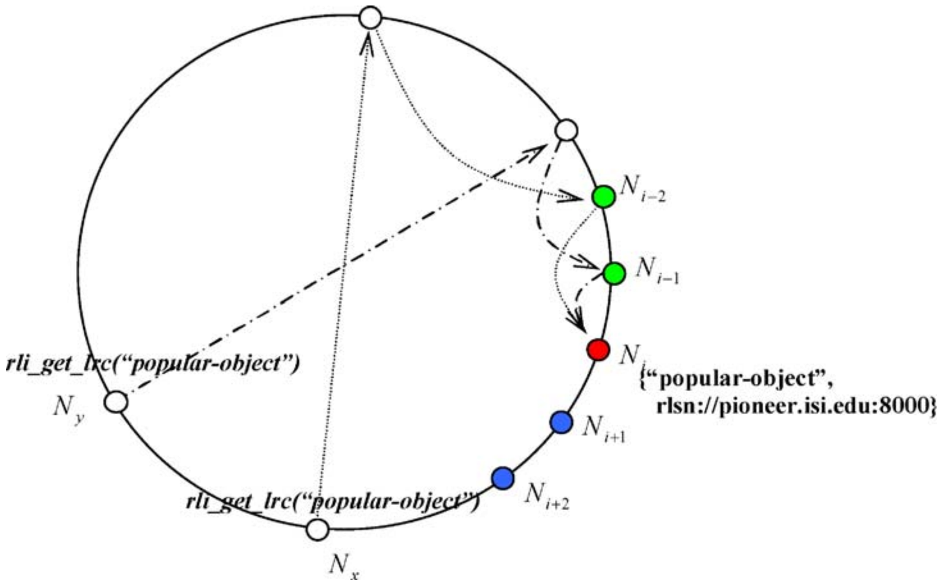
N_{i-2}
N_{i-1}
N_i
{“popular-object”, rlsn://pioneer.isi.edu:8000}
N_{i+1}
N_{i+2}
rli_get_lrc(“popular-object”)
N_y
rli_get_lrc(“popular-object”)
N_x

Figure 3. P-RLI Queries for logical name ‘popular-object’ traverse the predecessors of the root node N_i.

cause now each node stores its own mappings and those of its *k* successors.

5. P-RLS Implementation

We implemented the P-RLS system by extending the RLS implementation in Globus Toolkit 3.0 with Chord protocols. Figure 4 shows the architecture of our P-RLS implementation. In this implementation, each P-RLS node consists of an LRC server and a P-RLI server. The LRC server implements the same LRC protocol as the original RLS, but uses the Chord protocol to update {*logical name, LRC*} mappings. The P-RLI server implements both the 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 a *JOIN* message directly to the successor node to join the network. The *UPDATE* message is used to add or delete a *{logical name, LRC}* mapping, and the *QUERY* message is used to look up matching mappings for a specified logical name. The P-RLI nodes also periodically send *PROBING* and *STABILIZATION* messages to detect node failures and repair the overlay 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 sends 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 identifies the next hop, it forwards 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 to send the reply to the request originator directly. 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. We implemented the second approach to avoid too many open TCP connections. All LRC, RLI and Chord protocols are implemented on top of an RLS RPC layer called RRPC.

6. P-RLS Performance: Measurements, Analytical Models and Simulations

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.

6.1. *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 547 MHz processors with 1 GB of memory running

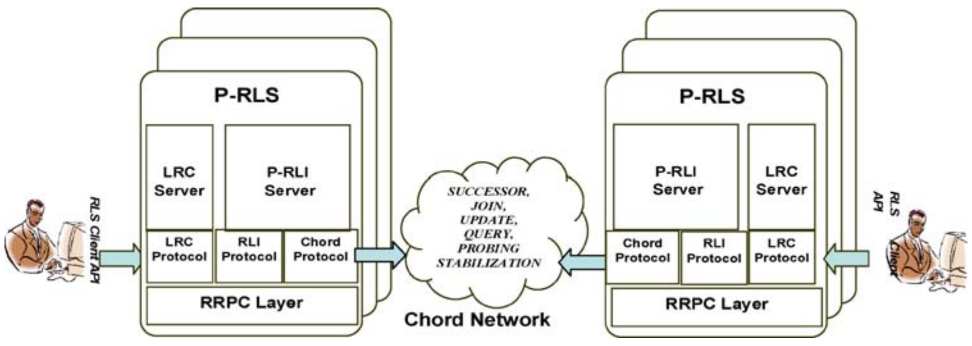

Figure 4. The P2P Replica Location Service Architecture.

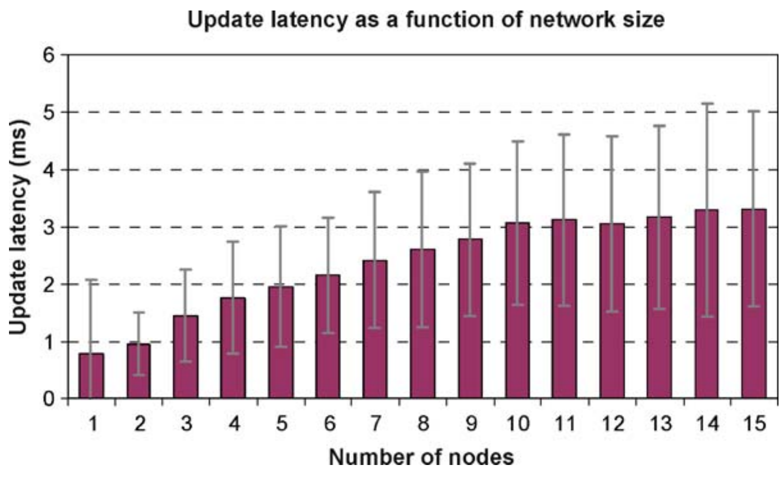

Figure 5. Shows update latency in milliseconds for performing an update operation in the P-RLS network.

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 1,000 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 1,000, with subsequent updates overwriting earlier ones for the same logical names. Similarly, the number of RPC calls required to perform these updates increases at a rate of $O(\log N)$ with the number of nodes in the system, N.

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 1,000 queries on each node, and the mean query latency and standard deviation are calculated. The results show that there is only a slight latency increase when we increase the number of mappings in the P-RLS network from 100,000 to 1 million. This is because the mappings on each node

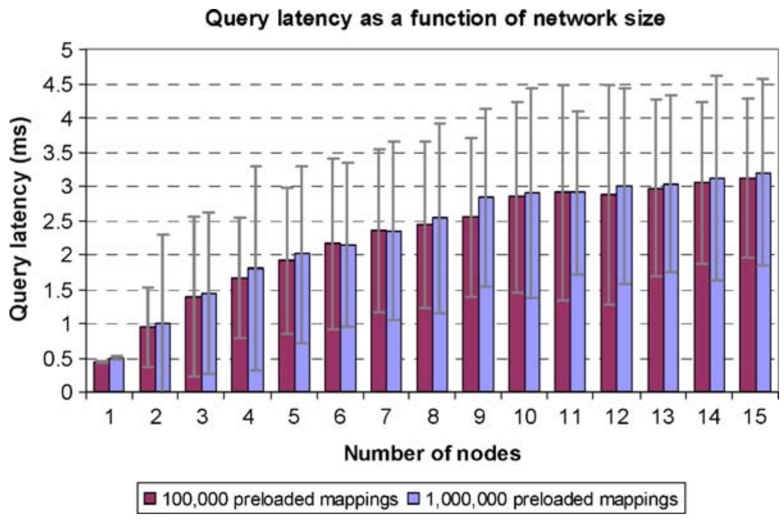

Figure 6. Shows update latency in milliseconds for performing an update operation in the P-RLS network.

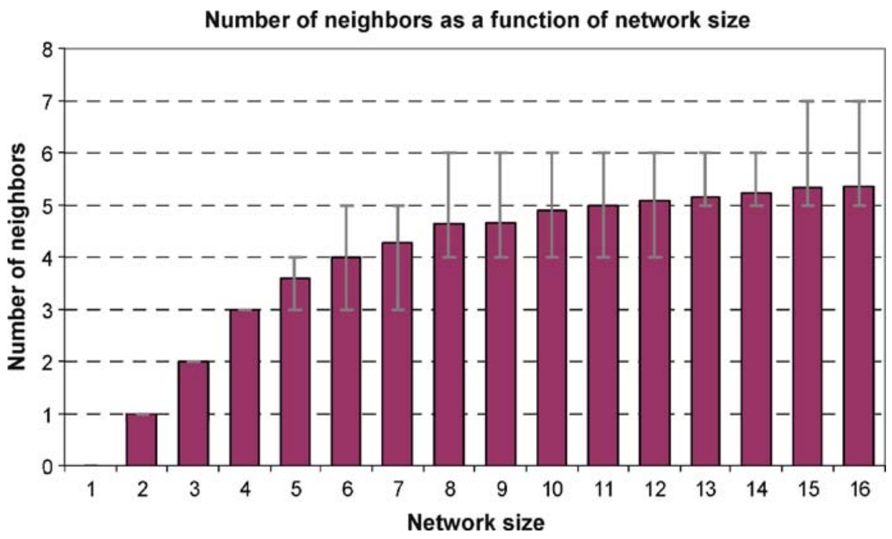

Figure 7. Rate of increase in pointers to neighbor nodes maintained by each P-RLI node as network size increases, where replication factor *k* is 2.

are stored in a hash table and the local lookup cost is nearly constant with respect to the number of mappings.

Because P-RLI queries are routed on average through *O* (log *N*) nodes in the P2P network, they have longer latency than a Globus RLI server, which can support up to 12,000 queries per second [19]. However, unlike the Globus RLS, the P-RLI network automatically routes queries to the correct P-RLI node and provides self-organization and self-healing.

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 7 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 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 and 10 s) at which the stabilization messages are sent. For both update intervals, the stabilization traffic is quite low (less than 1.5 KB/s 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, the Chord overlay network will react more quickly to node additions or failures. 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.

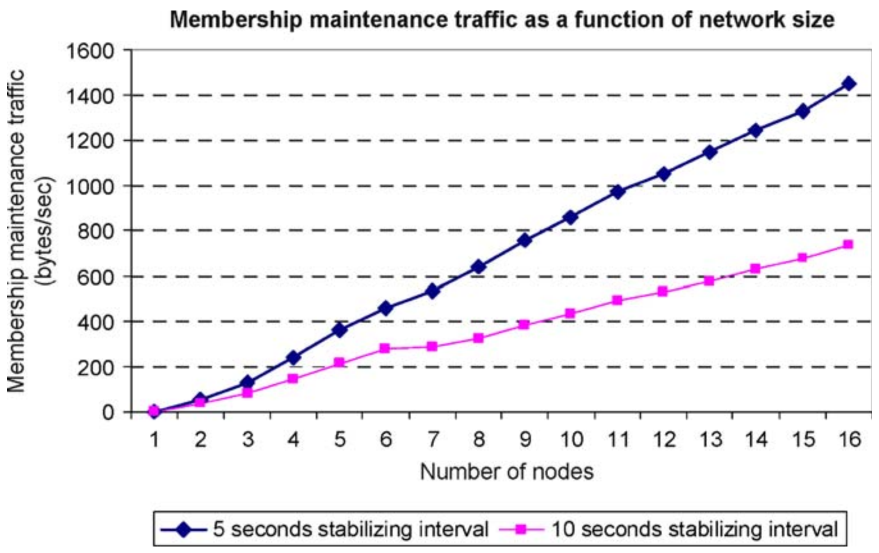

Figure 8. Stabilization Traffic for a P-RLS network of up to 16 nodes with stabilization intervals of 5 and 10 s.

6.2. *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.

Based on this analytical model, we computed the membership traffic for networks ranging from 10 to 10,000 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 8 of the previous section). Figure 9 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 1. Measured message sizes for our P-RLS implementation.

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

6.3. *Simulations for Adaptive Replication*

In this section, we present simulation results for a larger network of P-RLI nodes. We focus on the effect of successor replication in providing a more even distribution of mappings and the effect of predecessor replication in handling request hot spots.

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. We

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

	Stabilization interval	
Network size	5 s	10 s
10	876 bytes/sec	438 bytes/sec
100	14,533	7267
1000	203,077	101,538
10,000	2,608,190	1,304,095

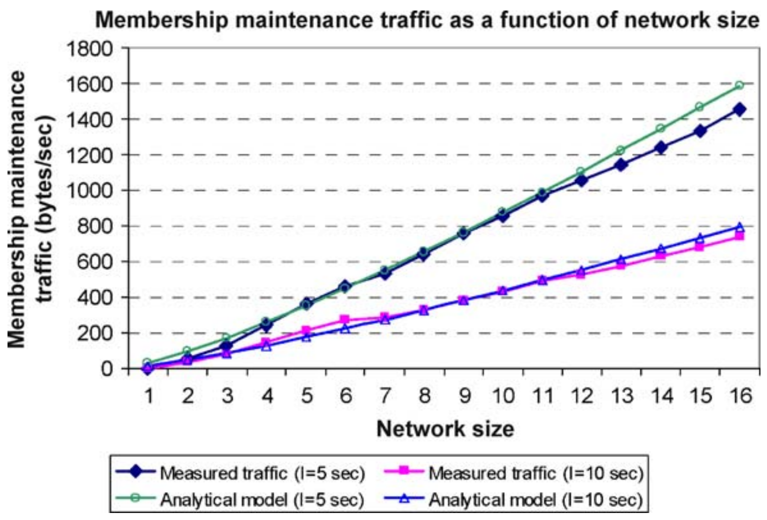

Figure 9. Comparison of measured and predicted values for stabilization traffic.

used different random seeds and ran the simulations 20 times.

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, where the RLS network contains 500,000 unique mappings. As the P-RLS network size increases, the average number of mappings per node decreases proportionally, while

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

	Replication factor (total replicas)			
Network size	0 (1)	1 (2)	4 (5)	12 (13)
10	50,000	100,000	250,000	N/A
100	5000	10,000	25,000	65,000
1000	500	1000	2500	6500
10,000	50	100	250	650

the mappings per node increase as the replication factor increases.

While 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. In Figure 10, we show that as the replication factor increases, mappings become more evenly distributed. The figure shows the distribution of mappings over a network of 100 P-RLI nodes. The horizontal axis shows the number of nodes ordered from the largest to the smallest number of mappings per node. The vertical axis shows 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 all 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, with a single replica, the 50% of nodes with the most mappings contain approximately 85% of all mappings, while for 13 total replicas, 50% of the nodes contain only about 60% of the total mappings.

Figure 11 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

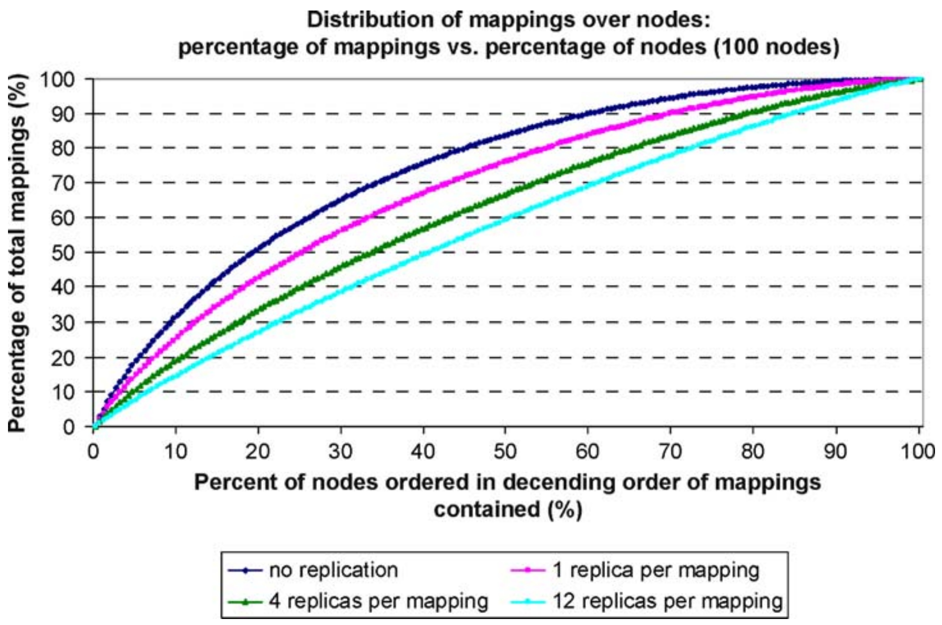

Figure 10. Shows cumulative percentage of total mappings stored *vs.* the percentage of total nodes for different replication factors, where nodes are ordered from highest to lowest mappings per node. There are 100 nodes in the P-RLS network and 500,000 unique mappings.

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.

Figures 10 and 11 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 sim-

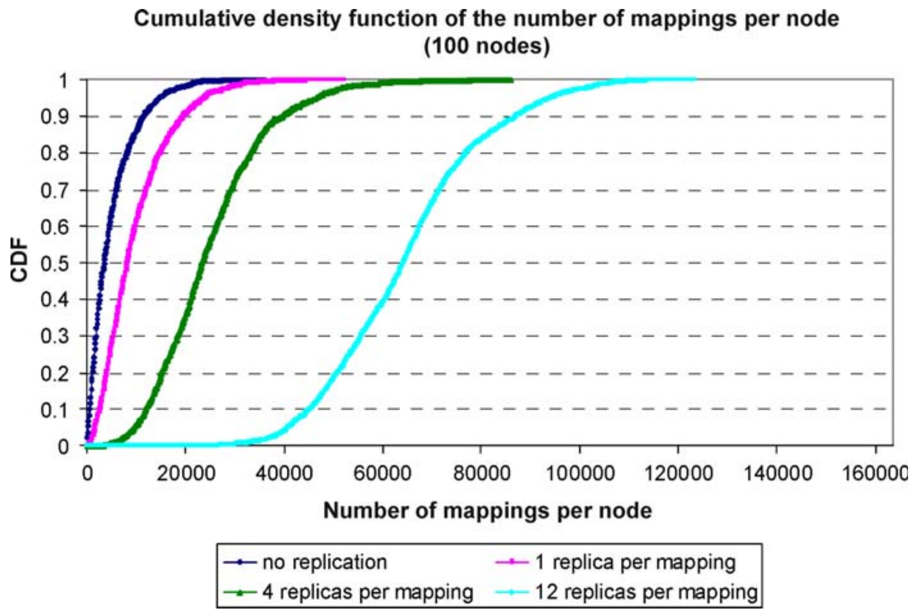

Figure 11. Cumulative distribution of mappings per node as the replication factor increases in a P-RLS network of 100 nodes with 500,000 unique mappings.

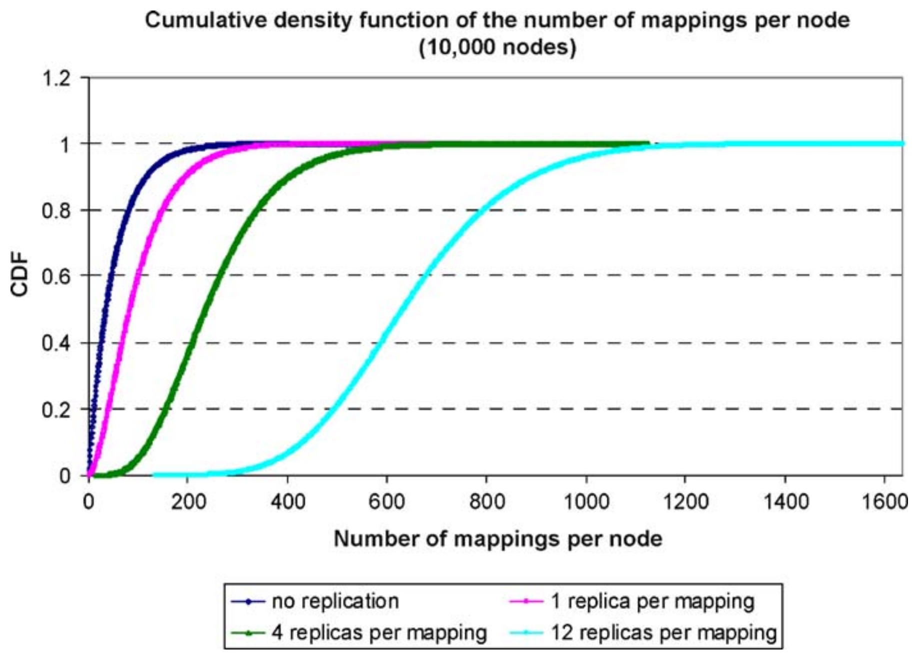

Figure 12. Cumulative density function for the number of mappings per node for a P-RLI network of 10,000 nodes for varying replication factors.

ilar results. Figure 12 shows a very similar cumulative distribution graph to that in Figure 11 for a network of 10,000 P-RLS nodes. The main difference between Figure 11 and Figure 12 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 networks.

Finally, we present simulation results for our predecessor replication scheme, which is designed to reduce query hotspots for popular mappings. Figure 13 shows simulation results for a P-RLS network with

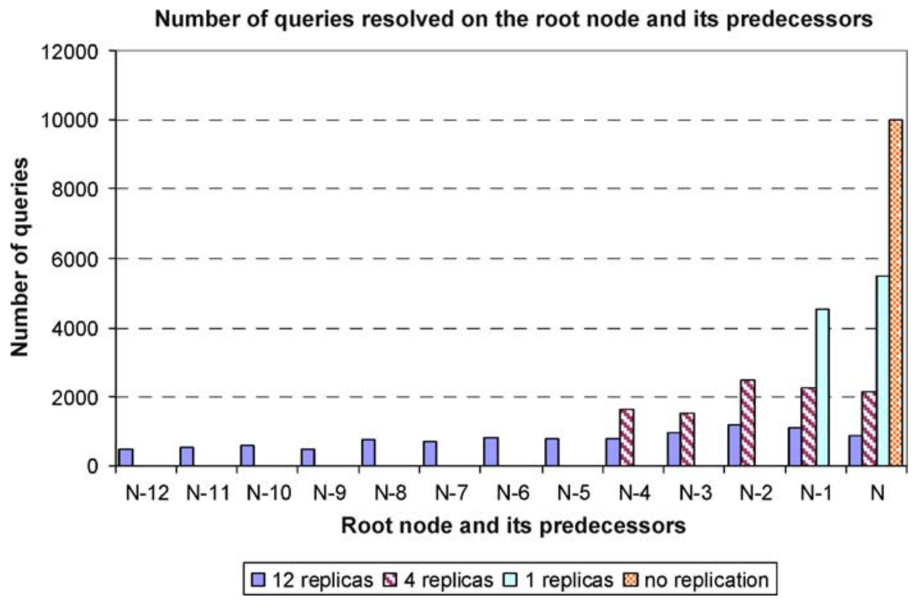

Figure 13. The number of queries resolved on the root node *N* and its predecessors is more evenly distributed as number of replicas per mapping increases.

10,000 nodes. We randomly choose 100 popular mappings. For each of these, we issue 10,000 queries from a randomly selected P-RLI node. We simulate the average number of queries that are resolved on the root node and its predecessors. In Figure 13, 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 *N* and its predecessors.

7. Oustanding Issues for Applying Peer-to-Peer Techniques to Grids

Peer-to-peer systems have attractive properties of self-configuration, self-healing, scalability and reliability. Incorporating peer-to-peer algorithms into Grid components therefore offers large potential benefits. However, Grid and peer-to-peer systems differ significantly. Peer-to-peer algorithms are likely to require modifications to function effectively in Grid resource discovery services.

7.1. *System Scale and Dynamism*

Peer-to-peer algorithms will likely change in Grid environments to reflect the different scale and dynamism exhibited by Grids. Peer-to-peer Internet file sharing algorithms have generally been optimized for environments that scale up to tens of thousands of participating nodes and with highly dynamic membership, in which nodes participate in the network for a few hours or days on average. By contrast, Grids are typically deployed at somewhat smaller scale, ranging from tens to hundreds and eventually thousands of components. In particular, the distributed resource discovery systems such as the Replica Location Service will likely have tens or hundreds of index services that need to self-organize and share information. Grids also exhibit less dynamism, with components remaining as members of a Grid for days, months or years.

One possible set of optimizations for peer-to-peer algorithms that takes advantage of the smaller scale and dynamism of Grid environments involves storing more information about neighbors in a structured peer-to-peer network. Since the total number of nodes in the network is relatively small, the additional storage overhead should also be small. This approach would reduce query latency by reducing the number of hops traversed during queries. The tradeoff of additional storage space consumed on nodes *versus* faster query speed may be practical at the scale of typical Grid but not practical for a much larger Internet peer-to-peer network.

7.2. *Security*

Security issues, particularly relating to authorization, have received less attention in peer-to-peer systems, but they are a key requirement for many Grids. In Internet file sharing applications such as Gnutella [16, 43], any node may join the peer-to-peer network, and access to content is not restricted. By contrast, Grids have strict requirements for mutual authentication, authorization and resource management. These requirements may apply both to nodes communicating with one another in a peer-to-peer discovery service and to clients querying the service.

In some environments, security mechanisms may not apply to the resource discovery service itself, but only to the underlying resources. For example, a peer-to-peer Replica Location Service might be available to all clients without restriction. Clients could query the system to discover the location of all data replicas, but clients might not have access permissions for all those replicas. Access control would be enforced by the storage system on which each replica resides.

In other environments, information about resource location will also be considered sensitive, and communication among peer-to-peer nodes of the resource discovery service as well as all client queries will require authentication and authorization.

7.3. *Structured vs. Unstructured Peer-to-Peer Systems*

In P-RLS, we have applied the structured Chord peer-to-peer overlay network to the Globus Replica Location Service. However, it is not clear in general whether structured or unstructured peer-to-peer networks are best-suited to Grid resource discovery services. While a great deal of research has been done on structured peer-to-peer networks based on distributed hash tables, the most successful and widely used Internet peer-to-peer file sharing systems, such as Gnutella, have all been unstructured. Chawathe et al. [17] have argued that unstructured systems are preferable for Internet file sharing because they are better able to handle the extremely transient nature of clients, the prevalence of keyword-based searches rather than exact-match queries, and the ability to answer queries for popular (and thus highly replicated) content quickly. It is an open question whether these same arguments for unstructured peer-to-peer networks apply to Grid resource discovery services, where the members of the distributed index are less transient, richer types of queries may be required, and applications may be less tolerant of inaccurate query results.

7.4. *Query Requirements*

Many existing peer-to-peer schemes support only simple queries, such as exact-match queries. However, many distributed resource discovery services in Grids such as information services need to support richer queries, including queries on multiple attributes, range queries on numerical attributes, or richer query languages, such as XPath. In addition, Grid services may support more complex data models, such as the GLUE (Grid Laboratory Uniform Environment) [22] information model that unifies resource information collected by different sources, or arbitrary XML, where the schema is known to the producer and consumer but not to the index components that store the information.

Another important issue is the effect on query performance of distributing index information in a structured peer-to-peer network. In a traditional hierarchical index, it is likely that information with similar names, such as related file names, will be stored on the same index server. An example of this is the Replica Location Service, where each Local Replica Catalog stores mappings for all the files on a local storage system. By contrast, in a structured peer-to-peer system using a distributed hash table, related mappings will tend to be distributed throughout the wide area network. The effect of this dispersal of related information on query performance has not yet been measured.

8. Additional Related Work

In Section 3, we provided an overview of work in peer-to-peer systems. In this section, we discuss additional related work in replica management for Grids and distributed file systems, and we discuss structured peer-to-peer systems other than Chord.

8.1. *Replica Management in Grids*

Other Grid systems for replica management include the Storage Resource Broker [9] and GridFarm [54] systems that register and discover replicas using a metadata catalog. These systems differ from RLS in several ways: They use a centralized catalog for replica registration and discovery; in addition to attributes related to physical replicas, they also maintain logical metadata information that describes the content of data files, which is deliberately kept separate in our system; and they use these metadata catalogs to maintain consistency among replicas, which is not done by RLS. The European DataGrid project [28] has implemented a different Replica Location Service based on the RLS Framework [18] that is used as part of their replica management architecture [32].

Ripeanu and Foster [46] constructed a peer-to-peer overlay network of Replica Location Services. Their decentralized, adaptive replica location mechanism uses three techniques: Bloom filter compression to create a *digest* or summary of the LFNs registered at a node, soft state update mechanisms, and a multicast overlay network. Unlike the P-RLS system, which uses a structured overlay to forward queries to a node that can answer an LFN query, this scheme distributes the Bloom filter digests to all nodes in the overlay. Thus, each node maintains a compressed image of the global system. When a client queries a node for a particular LFN mapping, the node first checks its locally stored mappings and answers the query, if possible. If not, the node checks its locally stored digests that summarize the contents of remote nodes. Finally, if the node finds a remote node whose digest contains a matching LFN, it contacts that node to obtain the mappings. Ripeanu et al. demonstrate that the network traffic generated using this approach is comparable to the traffic generated in query forwarding schemes. This solution requires more memory to store digests at a node, but query latencies are reduced compared to P-RLS because fewer network hops are required to resolve a query. 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.

8.2. *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 [14, 27, 37, 52, 55, 56]. 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 [36]. NFS [48] 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 [39] identifies volume locations by replicating a global mounting table among all sites. ASF [50] and Coda [58] employ a Volume Location Data Base (VLDB) for each local site and replicate it on the backbone servers of all sites. Ficus [36] 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.

8.3. *Structured Peer-to-Peer Networks*

Besides Chord, many other structured Peer-to-Peer networks have been proposed in recent years, such as Tapestry [60], Pastry [47], CAN [41], Koorde [30], Skip Graphs [8] and SkipNet [29].

The routing algorithms used in Tapestry and Pastry are both inspired by Plaxton [38]. 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 [15].

CAN [41] 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 two 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 $O(\log N/\log \log N)$.

Two structured P2P systems based on skip lists [40] have been proposed: Skip Graphs [8] and SkipNet [29]. 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 zero. 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.

9. Summary and Future Work

We have discussed issues for applying peer-to-peer techniques to Grid resource discovery services and have described the implementation and performance of the Peer-to-Peer Replica Location Service. Additional work planned for the P-RLS system includes measuring the performance of the system deployed in the wide area network, 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 are investigating applying peer-to-peer techniques to other Grid resource discovery services. In particular, we are applying an unstructured peer-to-peer overlay network to the index services of the Globus Monitoring and Discovery Service.

Acknowledgements

We are grateful to Mats Rynge for his technical support of our work; 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 Agreement DE-FC02-01ER25449 (SciDAC- DATA).

References

1. The Compact Muon Solenoid, An Experiment for the Large Hadron Collider at CERN, http://cmsinfo.cern.ch/Welcome.html/, 2005.
2. Gnutella, http://www.gnutella.com, 2004.
3. Grid and Utility Computing, http://devresource.hp.com/drc/topics/utility_comp.jsp, Hewlett Packard, 2004.
4. Grid Research Integration Deployment and Support Center, http://grids-center.org/, 2004.
5. QCDGrid: Probing the Building Blocks of Matter with the Power of the Grid, http://www.gridpp.ac.uk/qcdgrid/, 2005.
6. Southern California Earthquake Center (SCEC), http://www.scec.org/, 2005.
7. D.G. Andersen, H. Balakrishnan, M.F. Kaashoek and R. Morris, "The Case for Resilient Overlay Networks", in *8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Elmau/Oberbayern, Germany, 2001.
8. J. Aspnes, G. Shah, "Skip Graphs", in *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 2003.
9. C. Baru, R. Moore et al., "The SDSC Storage Resource Broker", in *CASCON'98 Conference*, 1998.
10. D. Bernholdt, S. Bharathi, D. Brown, K. Chancio, M. Chen, A. Chervenak, L. Cinquini, B. Drach, I. Foster, P. Fox, J. Garcia, C. Kesselman, R. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand and D. Williams, "The Earth System Grid: Supporting the Next Generation of Climate Modeling Research", *Proceedings of the IEEE*, Vol. 93, No. 3, pp. 485– 495.
11. G.B. Berriman et al., "Vol XXX, 2003, Montage a Grid Enabled Image Mosaic Service for the National Virtual Observatory", in *ADASS XIII, ASP Conference Series*, 2003.
12. R. Bhagwan, S. Savage and G.M. Voelker, "Understanding availability", in *The 2nd International Workshop on Peer-to-Peer Systems*, 2003.
13. B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors" *Communications of ACM*, Vol. 13, No. 7, pp. 422–426.

14. Y. Breitbart and H. Korth, "Replication and Consistency: Being Lazy Helps Sometimes", in *16th ACM SIGACT/ SIGMOD Symposium on the Principles of Database Systems*, Tucson, AZ, 1997.
15. M. Castro, P. Druschel, Y.C. Hu and A. Rowstron, "Topology-Aware routing in Structured Peer-to-Peer Overlay Networks", in *Intl. Workshop on Future Directions in Distributed Computing*, 2002.
16. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham and L. Breslau, "Making Gnutella-Like P2P Systems Scalable", in *ACM SIGCOMM 2003*, Karlshruhe, Germany, 2003.
17. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham and S. Shenker, "Making Gnutella-Like P2P Systems scalable", in *ACM SIGCOMM 2003*, Karlsruhe, Germany, 2003.
18. A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunst, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger and B. Tierney, "Giggle: A Framework for Constructing Scalable Replica Location Services", in *SC2002 Conference*, Baltimore, Maryland, 2002.
19. A.L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman and R. Schwartzkopf, "Performance and Scalability of a Replica Location Service", in *Thirteenth IEEE Int'l Symposium High Performance Distributed Computing (HPDC-13)*, Honolulu, Hawaii, 2004.
20. I. Clarke et al., "Protecting Free Expression Online with Freenet", *IEEE Internet Computing Journal*, Vol. 6, No. 1, pp. 40–49.
21. K. Czajkowski, S. Fitzgerald, I. Foster and C. Kesselman, "Grid Information Services for Distributed Resource Sharing", in *Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10)*, 2001, IEEE.
22. DataTag. Grid Laboratory Uniform Environment (GLUE), 2004.
23. E. Deelman et al., "Grid-Based Galaxy Morphology Analysis for the National Virtual Observatory", in *SC2003*, 2003.
24. E. Deelman et al., "Mapping Abstract Complex Workflows Onto Grid Environments", *Journal of Grid Computing*, Vol. 1, pp. 25–39.
25. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi and M. Livny, "Pegasus : Mapping Scientific Workflows Onto the Grid", in *Across Grids Conference*, Nicosia, Cyprus, 2004.
26. I. Foster and A. Iamnitchi, "On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing", in *Int'l Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, California, USA, 2003.
27. J. Gray, P. Helland, P. O'Neil and D. Shasha, "The Dangers of Replication and a Solution", in *ACM SIGMOD Conference*, 1996.
28. L. Guy, P. Kunszt, E. Laure, H. Stockinger and K. Stockinger, "Replica management in data grids", in *Global Grid Forum 5*, 2002.
29. N. Harvey et al., "SkipNet: A Scalable Overlay Network with Practical Locality Properties", in *Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, Washington, 2003.
30. F. Kaashoek and David R. Karger, "Koorde: A Simple Degree-Optimal Hash Table", in *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.
31. J.D. Kephart, "Chess The Vision of Autonomic Computing", *Computer Magazine*.
32. P. Kunszt et al., "Advanced Replica Management with Reptor", in *5th International Conference on Parallel Processing and Applied Mathematics*, Czestochowa, Poland, 2003, Springer.
33. LIGO Project, "LIGO – Laser Interferometer Gravitational Wave Observatory", http://www.ligo.caltech.edu/, 2004.
34. Q. Lv, P. Cao, E. Cohen, K. Li and S. Shenker, "Search and Replication in Unstructured Peer-to-Peer Networks", in *16th ACM International Conference on Supercomputing (ICS'02)*, New York, USA, 2002.
35. A.Y.C. Mizrak, V. Kumar and S. Savage, "Structured Superpeers: Leveraging Heterogeneity to Provide Constant-Time Lookup", in *IEEE Workshop on Internet Applications*, San Jose, 2003.
36. J.T.W. Page et al., "Management of Replicated Volume Location Data in the Ficus Replicated File System", in *USENIX Conference*, 1996.
37. K. Petersen et al., "Flexible Update Propagation for Weakly Consistent Replication", in *16th ACM Symposium on Operating Systems Principles (SOSP-16)*, Saint Malo, France, 1997.
38. C. Plaxton, R. Rajaraman and A. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment", in *ACM SPAA*, Newport, Rhode Island, 1997.
39. G. Popek, *The Locus Distributed System Architecture*, MIT, 1986.
40. W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", in *Workshop on Algorithms and Data Structures*, 1989.
41. S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A Scalable Content-Addressable Network", in *ACM SIGCOMM*, 2001.
42. S. Ratnasamy, S. Shenker and I. Stoica, "Routing Algorithms for DHTs: Some Open Questions", in *IPTPS02*, Cambridge, USA, 2002.
43. M. Ripeanu, "Peer-to-Peer Architecture Case Study: Gnutella Network", in *IEEE 1st International Conference on Peer-to-peer Computing (P2P2001)*, Linkoping, Sweden, 2001, IEEE.
44. 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*, Vol. 6.
45. 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*.
46. M. Ripeanu and Ian Foster, "A Decentralized, Adaptive, Replica Location Mechanism", in *11th IEEE International*

Symposium on High Performance Distributed Computing (HPDC-11), Edinburgh, Scotland, 2002.

47. A. Rowstron and P. Druschel, "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems", in *International Conference on Distributed Systems Platforms (Middleware)*, 2001.
48. R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, "Design and Implementation of the Sun Network File System", in *USENIX Conference*, 1985.
49. S. Saroiu, P.K. Gummadi and S.D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems", in *Multimedia Computing and Networking*, 2002.
50. M. Satyanarayanan et al., "Coda: A Highly Available System for a Distributed Workstation Environment", *IEEE Transactions on Computers*, Vol. 39, No. 4, pp. 447–459.
51. S. Sen and Jia Wong, "Analyzing Peer-to-Peer Traffic Across Large Networks", in *Proceedings of the Second ACM SIGCOMM Workshop on Internet Measurment*, 2002.
52. J. Sidell et al., "Data Replication in Mariposa", in *12th International Conference on Data Engineering*, New Orleans, Los Angeles, 1996.
53. I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications", in *ACM SIGCOMM*, 2001.
54. O. Tatebe et al., "Worldwide Fast File Replication on Grid Datafarm", in *2003 Computing in High Energy and Nuclear Physics (CHEP03)*, 2003.
55. D.B. Terry, K. Petersen, M.J. Spreitzer and M.M. Theimer, "The Case for Non-transparent Replication: Examples from Bayou", in *14th International Conference on Data Engineering*, 1998.
56. M. Wiesmann et al., "Database Replication Techniques: A Three Paramater Classification", in *19th IEEE Symposium on Reliable Distributed Systems*, Nuernberg, Germany, 2002.
57. B. Yang and H. Garcia-Molina, "Designing A Super-Peer Network", in *IEEE Int'l Conf. on Data Engineering*, 2003.
58. E.R. Zayas and C.F. Everhart, "Design and Specification of the Cellular Andrew Environment", Carnegie–Mellon University, 1988.
59. X. Zhang, J. Freschl and J.M. Schopf, "A Performance Study of Monitoring and Information Services for Distributed Systems", in *Twelfth IEEE Int'l Symposium High Performance Distributed Computing (HPDC-12)*, Seattle, Washington, 2003.
60. B.Y. Zhao, J.D. Kubiatowicz and A.D. Joseph, "Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing", U.C. Berkeley, 2001.