The Globus Replica Location Service: Design and Experience

Ann L. CHERVENAK, Robert SCHULER, Matei RIPEANU, Muhammad Ali AMER, Shishir BHARATHI, Ian FOSTER, Adriana IAMNITCHI, Carl KESSELMAN

***Abstract*-- Distributed computing systems employ replication to improve overall system robustness, scalability and performance. A Replica Location Service (RLS) offers a mechanism to maintain and provide information about physical locations of replicas. This paper defines a design framework for replica location services that supports a variety of deployment options. We describe the RLS implementation that is distributed with the Globus Toolkit and is in production use in several Grid deployments. Features of our modular implementation include the use of soft-state protocols to populate a distributed index and Bloom filter compression to reduce overheads for distribution of index information. Our performance evaluation demonstrates that the RLS implementation scales well for individual servers with millions of entries and up to one hundred clients. We describe the characteristics of existing RLS deployments and discuss how RLS has been integrated with higher-level data management services.**

***Index Terms*-- Replica Location Service, Replica Management, Grids, Data Management**

I. INTRODUCTION

Data management in Grid environments is a challenging problem. Data-intensive applications may produce data sets on the order of terabytes or petabytes that consist of millions of individual files. These data sets may be replicated at multiple locations to increase storage reliability, data availability, and/or access performance. For example, scientists performing simulation or analysis often prefer to have local copies of key data sets to guarantee that they will be able to complete their work without depending on remote sites or suffering wide-area access latencies.

The Replica Location Service (RLS) [1, 2] is one component of an overall Grid data

Manuscript received December 20, 2006. This work was supported in part by the U.S. Department of Energy under DOE Cooperative Agreement DE-FC02-01ER25449 (SciDAC- DATA) & DE-FC02-01ER25453 (SciDAC-ESG).

Ann L. Chervenak, Robert Schuler, Muhammad Ali Amer, Shishir Bharathi and Carl Kesselman are with the University of Southern California Information Sciences Institute, Marina del Rey, CA 90292 USA (telephone: 310-822-1511, e-mail: annc, schuler, mamer, shishir, carl@isi.edu).

Matei Ripeanu is with the Department of Electrical and Computer Engineering Department of the University of British Columbia, Vancouver, BC, BC V6T 1Z4, Canada (e-mail matei@ece.ubc.ca).

Ian Foster is with Computation Institute of Argonne National Laboratory and the University of Chicago, Chicago, IL 60637 USA (e-mail: foster@mcs.anl.gov).

Adriana Iamnitchi is with the Computer Science and Engineering Department of the University of South Florida, Tampa, FL 33620 USA (email: anda@cse.usf.edu).

management architecture. The RLS provides a mechanism for registering the existence of replicas and for discovering them. We have designed and implemented a distributed RLS that consists of two components: a Local Replica Catalog (LRC) that stores mappings from logical names of data items to their addresses on a storage system and a Replica Location Index (RLI) that contains information about the mappings stored in one or more LRCs and answers queries about their contents. Our RLS server implementation performs well in Grid environments, scaling to register millions of entries and supporting up to one hundred simultaneous clients.

Grid data management schemes typically integrate the RLS with other components to offer higher-level capabilities including creation, discovery, maintenance, and access to replicated data. These components include *data transfer* services used to create and access replicas, *consistency* services that propagate updates to replicas and maintain a specified degree of consistency, *verification* services that verify the correctness of registered replicas, and *selection* services that choose among replicas based on the current state of Grid resources.

The RLS is in production use in a variety of Grid scientific environments, including the Lightweight Interferometer Gravitational Wave Observatory (LIGO) project [3], the Earth System Grid (ESG) [4, 5], the Quantum Chromodynamics (QCD) Grid [6], and the Southern California Earthquake Center [7]. The largest production deployment to date is the LIGO collaboration, which registers mappings from over 25 million logical files to over 120 million physical files distributed across ten sites. The RLS has been integrated into several higher-level, production data management systems, including the LIGO Lightweight Data Replicator [8], the ESG web portal [4, 5] and the Pegasus planning and workflow management system [9].

This paper presents the design of the Replica Location Service (Sections II, III and IV), its implementation (Section V) and performance (Section VI). It also describes how RLS is

currently used in production scientific environments (Section VII). Finally, we present ongoing efforts (Section VIII) and related work (Section IX).

II. The RLS Framework

The Replica Location Service design and implementation are based on a framework [2] jointly developed by the Globus and European DataGrid projects. This section presents the five elements of the RLS framework: Local Replica Catalogs, Replica Location Indexes, soft-state update mechanisms, optional compression of updates, and membership services.

A. The Local Replica Catalog

Local Replica Catalogs (LRCs) maintain mappings between logical names and target names. Logical names are unique identifiers for data items that may have one or more physical replicas. Logical namespaces are application-specific, and tools are available for generating unique identifiers [10]. Target names are typically the physical locations of data, but they may also be logical names. An LRC implementation may associate attributes with logical or target names.

Clients query LRC mappings to discover target names associated with a logical name and the reverse. Clients also query for logical or target names with specified attribute values, where attributes are arbitrary name-value pairs defined by an RLS configuration.

A Local Replica Catalog may periodically send updates that summarize its state to one or more Replica Location Indexes (RLIs). In addition, an LRC maintains security over its contents, performing authentication and authorization of users via standard Grid mechanisms.

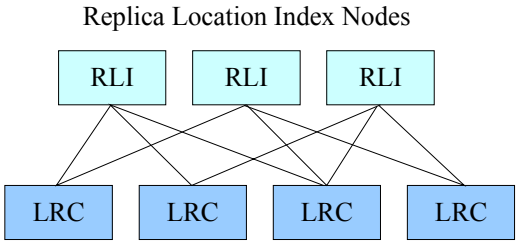


Figure 1: Hierarchical RLS configuration

B. The Replica Location Index

Each RLS deployment may contain one or more Replica Location Indexes (RLIs) that store and answer queries about mappings held in one or more LRCs. Figure 1 illustrates an RLS deployment in which four LRCs send state updates to three RLIs.

An RLI stores mappings from logical names to the LRCs that, in turn, contain mappings from those logical names to target names. Each RLI responds to queries about logical names, providing a list of one or more LRCs believed to contain target mappings associated with that logical name. RLIs accept periodic soft-state updates (described below) from one or more LRCs that summarize LRC state. An RLI provides security, performing authentication and authorization operations using standard Grid mechanisms.

Section IV illustrates a variety of RLI index structures with different performance and reliability characteristics determined by the number of RLIs and the amount of redundancy in the index. The RLS framework also supports partitioning of updates from an LRC server among RLI servers to reduce the amount of information sent to an RLI. Options include partitioning based on the logical namespace, with updates for portions of the namespace sent to different RLIs, and partitioning based on the domain namespace of target names.

C. Soft State Updates

Information in the distributed RLIs is maintained using *soft-state update protocols*. Soft state information times out and must be periodically refreshed. Soft state update protocols provide two advantages. First, they decouple the state producer and consumer; the state consumer removes stale data implicitly via timeouts rather than using an explicit protocol that requires communication with the producer. Second, state consumers need not maintain persistent state; if a consumer fails and later recovers, its state can be reconstructed using soft state updates. In our design, each LRC sends soft state information about its mappings to one or more RLIs.

D. Compression of Soft-State Updates

Soft-state updates may optionally be *compressed* to reduce the amount of data sent from LRCs to RLIs and to reduce storage and I/O requirements on RLIs. The RLS framework proposes two compression options. The first option reduces the size of soft state updates by taking advantage of structural or semantic information present in logical names, for example, by sending information about collections of related logical names rather than individual logical names. A second option is to use a hash digest technique such as Bloom filter compression [11].

While compression schemes based on hash digests greatly reduce network traffic and provide better scalability, they do not support wildcard queries. This limitation, however, is not a major concern in practice for the RLS, since many data grid deployments include an additional service, a *metadata catalog*, that supports queries based on metadata attributes. These queries identify a list of logical files with specified attributes. The RLS is then used to locate physical replicas of these logical files. Thus, wildcard queries are typically not required at the RLS level, since specific logical file names are identified by the metadata catalog. This organization supports a separation of concerns: the metadata catalog provides a flexible metadata definition and an expressive query interface, and RLS provides scalability and throughput for location operations.

E. Membership Management for RLS Services

The final component of the RLS framework is a *membership service* that tracks the set of participating LRCs and RLIs and responds to changes in membership, for example, when a server joins or leaves the RLS. Existing RLS production deployments use a simple, manually-maintained, static configuration for RLS servers that must be modified by an administrator to change update patterns when servers enter or leave the system. Ideally, membership management

would be automated so that LRCs and RLIs can discover one another and updates are redistributed to balance load when servers join or leave the system. More automated membership management alternatives include peer-to-peer self-organization of servers [12, 13] and a service registry that maintains a list of active servers and facilitates reconfiguration.

III. RLS AND CONSISTENCY

The RLS does not provide consistency guarantees between its mappings and the contents of file systems. It does not verify that physical files that are registered as replicas are actually copies of one another. If registered replicas are modified so that they are no longer valid copies of one another, the RLS will not detect these changes or take action. While such validation checks are desirable in some environments, they would also add significant overhead and complexity to the RLS. Our implementation relies on higher-level data management or consistency services to perform validation checks when necessary.

There are several reasons for this design choice. One is to provide a simple and efficient registry that avoids the overhead and performance limitations associated with maintaining strong consistency across wide area systems, e.g., by calculating and comparing checksums for replicas. Second, providing a high level of consistency is not required for many applications. Scientific applications often publish data that is accessed in a read-only mode, so consistency checking is not required. Often these applications have a small set of trusted users who publish data in bulk and cannot tolerate overheads associated with performing consistency checking operations on every published data item. Third, applications may have different definitions of what constitutes a “valid” replica. For some applications, replicas are exact byte-for-byte copies of one another. For others, files may be considered replicas if they are different versions of the same file or have

the same content in different formats (e.g., compressed and uncompressed). We do not prescribe a particular definition for replicas in RLS. For these reasons, we leave consistency checking operations to higher-level services, an approach also taken by Amazon's Dynamo service [14].

IV. RLS Configuration Options

The performance, scalability and reliability of RLS are determined by the number of servers deployed, the configuration of updates from LRCs to RLIs, and the scalability of the database back end. This section outlines an array of deployment options and discusses their tradeoffs.

The simplest RLS deployment uses a single, centralized Local Replica Catalog that registers mappings for replicas stored at one or more sites. While a centralized RLS is simple to deploy and maintain, it may not scale well with the number of mappings or the number of simultaneous clients. A centralized deployment also represents a single point of failure. For these reasons, a variety of distributed RLS configurations may provide better reliability or scalability.

In another configuration option, multiple LRCs update a single, centralized RLI index, as shown in Figure 2(a). In this case, there is a single location where all RLI queries are directed, and the RLI is aware of all mappings in LRCs. However, this RLI represents a single point of failure in the system and may be a performance bottleneck. We can improve reliability and load balancing by configuring the system with redundant RLI index servers, where each RLI receives updates from every LRC, as in Figure 2(b). Since the RLIs receive the same update information, either can respond to RLI queries.

Current RLS deployments typically integrate a relatively small number of sites. For example, the Earth System Grid [4, 5] deploys RLS servers at five sites, while the LIGO physics collaboration [3, 15] deploys servers at ten sites. For deployments of this scale, a fully connected

RLS configuration (Figure 2(c)) is a reliable option. A fully connected RLS deploys an LRC and an RLI at each site, and all LRCs send updates to all RLIs, employing a "write-all read-any" strategy [16]. Advantages of this configuration include a high degree of redundancy for the RLI index. Also, each RLI has information about all replicas in the system.

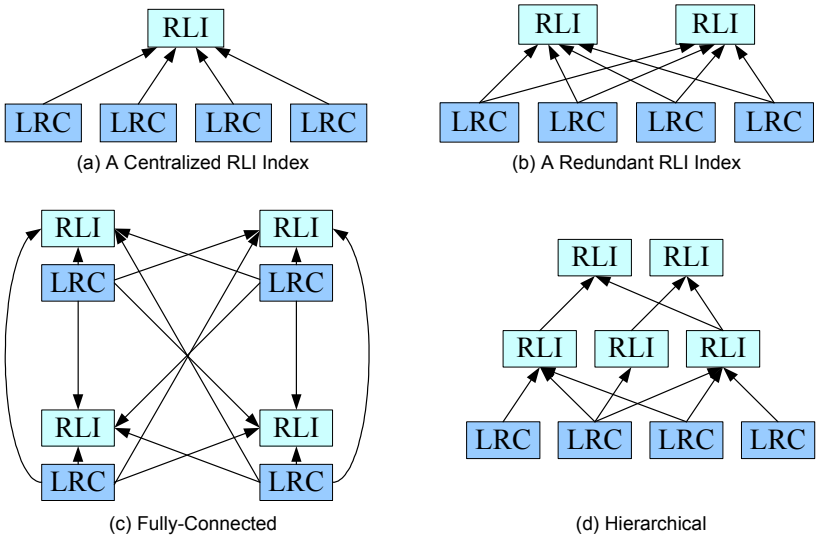


Figure 2: RLS Configuration Options

For larger deployments, a fully-connected RLS configuration would generate high traffic volume for updates. To reduce this traffic, the RLS can be partitioned so that an RLI receives updates from a subset of LRCs. In addition, the RLS can be configured with a hierarchy of RLI servers (Figure 2(d)), where RLIs receive updates from other RLIs and summarize their contents.

For increased scalability, we have also examined peer-to-peer organizations of RLS servers (described in Section VIII.A), but these are not supported in the current RLS implementation.

V. The Globus RLS Implementation

Based on the RLS framework above, we have implemented a Replica Location Service that has been included in the Globus Toolkit starting with version 3.0. This section describes the features of this implementation. The current implementation does not include a membership service, but instead uses a simple static configuration of LRCs and RLIs. We are investigating mechanisms for automatic configuration of RLS servers, including peer-to-peer techniques [12, 13].

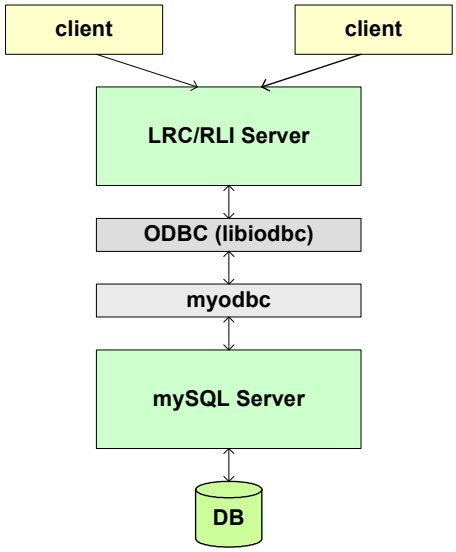


Figure 3: The RLS Implementation

A. The Common LRC and RLI Server

Although the RLS framework treats the LRC and RLI servers as separate components, our implementation (shown in Figure 3) consists of a common server that can be configured as an LRC, an RLI, or both.

The RLS server is multi-threaded and is written in C. The server supports Grid Security Infrastructure (GSI) authentication. One level of authorization is provided using an optional Globus *gridmap* file that contains mappings from Distinguished Names (DNs) in users' X.509 certificates to local usernames; if a DN is not included in the *gridmap* file, then access is denied. The RLS server can also be configured with an access control list that specifies the RLS operations each user is authorized to perform. Access control list entries are regular expressions that grant privileges such as *lrc_read* and *lrc_write* access to users based on either the DN in the user's certificate or based on the local username specified by the *gridmap* file. The RLS server can also be run without authentication or authorization controls.

The RLS server back end is a relational database accessed through an Open Database Connectivity (ODBC) layer. RLS interoperates with MySQL, PostgreSQL, SQLite and Oracle databases. The LRC database structure includes a table for logical names, a table for target names, and a mapping table that provides associations between logical and target names. There is also a general attribute table that associates user-defined attributes with either logical names or target name as well as individual tables for each attribute type (string, int, float, date). Attributes are often used to associate such values as size with a physical name for a file or data object. Finally, one table lists all RLIs updated by the LRC, and another table stores regular expressions

for LRC namespace partitioning.

The RLI server uses a relational database back end when it receives full, uncompressed updates from LRCs. This relational database contains three simple tables: one for logical names, one for LRCs and a mapping table that stores {LN, LRC} associations. Soft state updates with Bloom filter compression, when used, are stored in memory.

B. Soft State Updates

Local Replica Catalogs send periodic summaries of their state to Replica Location Index servers. In our implementation, soft state updates may be of four types: uncompressed updates, incremental updates, and updates using Bloom filter compression or name space partitioning.

Soft state information eventually expires and must be either refreshed or deleted. An *expire* thread runs periodically and examines timestamps in the RLI mapping table, discarding entries older than the allowed timeout interval.

With periodic updates, there is a delay between when LRC mappings change and when those changes are propagated to RLIs. Thus, an RLI query may return stale information, and a client may not find a mapping for a desired logical name when it queries an LRC. Applications must be sufficiently robust to recover from this situation and query for another replica.

Uncompressed updates: An uncompressed soft state update contains a list of all logical names for which mappings are stored in an LRC. The RLI creates associations between these logical names and the LRC. To discover one or more target replicas for a logical name, a client queries an RLI, which returns pointers to zero or more LRCs that contain mappings for that name. The client then queries LRCs to obtain target name mappings.

Incremental Updates: To reduce the frequency of full soft state updates and the staleness of RLI information, our implementation supports a combination of infrequent full updates and more

frequent incremental updates that reflect recent changes to an LRC. Incremental updates are sent after a short, configurable interval has elapsed or after a specified number of LRC updates have occurred. Periodic full updates are required to refresh RLI information that eventually expires.

Compressed Updates: Our implementation includes a Bloom filter [11] compression scheme. An LRC constructs a Bloom filter (a bit array) that summarizes LRC state by computing multiple hash functions on each logical name registered in the LRC and setting the corresponding bits in the Bloom filter. The resulting bitmap is sent to an RLI, which stores one Bloom filter in its memory per LRC. (The RLI may also store Bloom filters on disk for fast recovery after reboots.) When an RLI receives a query for a logical name, it computes the same hash functions against the logical name and checks whether the corresponding bits in each LRC Bloom filter are set. If any of the bits is not set, then the logical name was not registered in the corresponding LRC. If all the bits are set, then that logical name has likely been registered in the corresponding LRC. There is a small possibility, however, of a *false positive*, i.e., a false indication that the LRC contains the mapping. The probability of false positives is controlled by the size of the Bloom filter and the number of hash functions used. Our implementation sets the size of the Bloom filter based on the number of mappings in an LRC (e.g., 10 million bits for 1 million entries) and uses three hash functions by default. These parameters give a false positive rate of approximately 1%.

Partitioning: Finally, our implementation supports partitioning of updates based on the logical name space. When partitioning is enabled, logical names are matched against regular expressions, and updates relating to different subsets of the logical namespace are sent to different RLIs. The goal of partitioning is to reduce the size of soft state updates between LRCs and RLIs. We have observed that, in practice, partitioning is rarely used, because users consider that Bloom filter compression offers sufficient performance and efficiency.

C. RLS Clients

The RLS implementation includes interfaces written in C, Java, and python as well as a command line client tool. We have also implemented a Web service interface to RLS that is compatible with the Web Services Resource Framework (WS-RF) standards [17].

TABLE 1
RLS OPERATIONS

LRC Operations	
Mapping management	Create, add, delete, rename mappings for single entities and in bulk
Attribute management	Create, add, modify, delete single attributes and in bulk
Query operations	Query based on logical or target name, wildcard queries, bulk queries, query based on attribute names or values
LRC management	Query RLIs updated by this LRC, add RLI to update, remove RLI from update list
RLI Operations	
Query operations	Query based on logical name, bulk queries, wildcard queries
RLI management	Query LRCs that update RLI

Table 1 lists many of the operations provided by the LRC and RLI clients. Each of these operations may correspond to multiple SQL operations on back end database tables.

D. RLS Bulk Operations

For user convenience and improved performance, the RLS implementation includes bulk operations that bundle together a large number of *add*, *query*, and *delete* operations on mappings and on attributes. Bulk operations are particularly convenient for scientific applications that perform many RLS *query* or *update* operations at the same time, for example, when publishing or processing large data sets. Bulk operations are essential for high performance in RLS, since many operations can be submitted to the catalog while incurring the overhead of a single client request. Thus, bulk operations avoid the overhead of issuing each request separately.

VI. PERFORMANCE AND SCALABILITY OF THE RLS IMPLEMENTATION

This section presents a performance and scalability study of the Globus RLS implementation. The software versions used in this study are indicated in Table 2.

TABLE 2
SOFTWARE VERSIONS USED IN EVALUATION

Replica Location Service	v4.3
Globus Packaging Toolkit	v3.2
libiODBC library	v3.0.5
MySQL database	v5.0.21
MyODBC library	v3.51.06

A. Overall RLS Performance

Our first set of tests evaluates the overall performance of LRCs. We present operation rates for *query*, *add* and *delete* operations. Requests are submitted by multi-threaded client programs written in Java that specify the number of threads that submit requests to a server and the operations to perform (query, add or delete mappings). Our testbed includes up to 20 machines that run five client threads each. Each client thread performs thousands of operations by issuing one or more bulk requests. We compute the operation rate (in queries per second) from the total operations performed by client threads over time. Each data point in the graphs below shows the average operation rates over 5 different trials, with error bars indicating the standard deviation.

In the experiments shown in this paper, the number of operations performed per client per trial ensures that we measure sustained peak operation rates for the designated number of clients running in parallel. We define the *warmup time* as the interval between the time of the first operation issued by the first client to the time of the first request issued by the last client. *Cooloff time* is defined as the interval between the time when the first client finishes submitting operations to the time the last client finishes submitting requests. The remaining time, when all clients are submitting requests in parallel, is the *overlap time*. In all our experiments, we ensure that the warmup time and the cooldown time represent a relatively small percentage of the overall execution time. In these experiments, a minimum of 78.6% of the measured execution time is overlap time. Figure 5 shows an overlap profile for a query experiment.

For each trial, a server is loaded with 1 million base mappings. We keep the database size relatively constant during all tests to avoid performance variations due to changes in database

size. For example, in add/delete tests, the mappings that are added in each trial are deleted before subsequent trials are performed. We limit the variation in the size of the database during a trial to no more than 20% of the original size.

The second set of experiments evaluates the performance of soft state updates between LRC and RLI servers. We measure the performance of uncompressed updates and updates that use Bloom filter compression. For these tests, LRC servers are loaded with mappings and forced to update an RLI server. Update time is measured from the LRC's perspective.

Our experiments are executed on the following systems. The client machines are nodes of a Linux cluster: dual-processor Pentium-III 547 MHz with 1.5 GB of memory running the RedHat v.9 operating system. For experiments measuring RLS server query and update performance, the client machines run an RLS client program. For experiments measuring soft state update performance, the client machines run LRC servers. Server machines are dual-processor Intel Xeon 2.2 GHz workstations with 1 GB of memory running either the RedHat v8 or Debian 3.0. The server at Argonne National Laboratory used for the remote RLI experiments runs on a dual processor Intel Xeon 2.2 GHz machine with 2 GB of memory running Red Hat 7.3.

For all experiments, the back end database used is MySQL using the default configuration in which updates are buffered and not immediately flushed to disk. This option provides high update performance at the cost of possible inconsistencies in case of machine failures [1].

Except for one query comparison test, GSI authentication is enabled on the RLS server. In all experiments, RLS logging is disabled.

Figure 4 compares the operation rates achieved by issuing queries to an LRC with those achieved for queries to a native java MySQL database using an ODBC client. For the LRC, we show bulk query performance with and without authentication as well as the performance of non-

bulk RLS queries. Up to 20 client machines with five threads per client issue operations to a single LRC. For an experimental trial, each client performs a total of 15,000 query operations by issuing 15 bulk query requests with 1000 queries per bulk request. Each data point in the graph shows the average query rate over 5 trials, with error bars indicating the standard deviation.

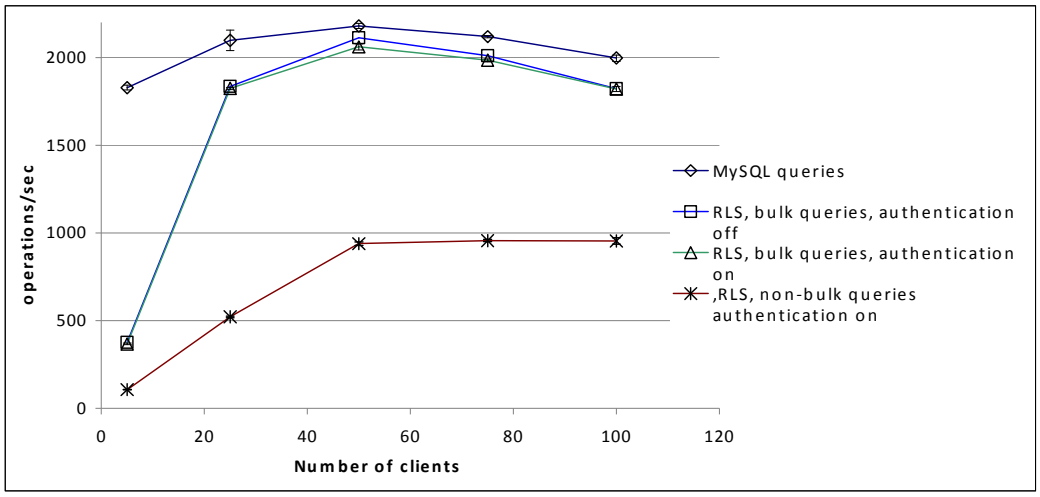


Figure 4: Comparison of query performance for RLS (with and without GSI authentication) and native ODBC client for MySQL

The LRC achieves a bulk query rate of almost 2000 per second. The graph shows that RLS overhead compared to the native MySQL query rates is 7.3 % on average for 25 to 100 clients and 79.4% for 5 clients. The overhead of performing authentication on RLS operations is at most 2.5%. The graph also shows the advantage of using bulk operations, which achieve approximately double the query rate of non-bulk operations. This difference is due to the overhead of issuing requests and establishing database connections for each non-bulk query call, compared to incurring those overheads once for a bulk operation. Bulk operations are performed individually at the LRC server. The mean standard deviation for each of these averaged data points, shown in error bars, is 0.86%.

Figure 5 shows the overlap profile for the measurements of LRC bulk query rates with authentication from Figure 4. The minimum measured overlap is 78.6% for 25 clients.

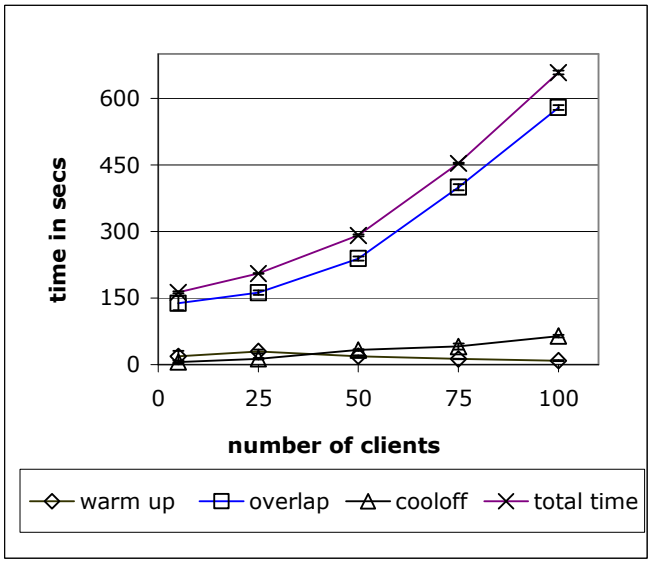


Figure 5: Overlap profile

Figure 6 shows RLS performance for bulk add and delete operations compared to that of an ODBC client that performs the same sequence of database operations. The ODBC client issues all operations on a per transaction basis, rather than through bulk calls. For RLS measurements, we perform a constant number of 300,000 operations per trial. Thus, for 100 clients we perform 3000 operations per client, for 50 clients we perform 6000 operations per client, etc. We compare these numbers with a native ODBC client that performs 3000 operations per client, which provides sufficient overlap. LRC bulk add rates range from 9.5% lower than native non-bulk add rates for 25 clients to 15.5% higher for 100 clients. During these tests, the ODBC client experienced deadlocks. RLS bulk delete rates are 28% lower on average than native ODBC delete rates.

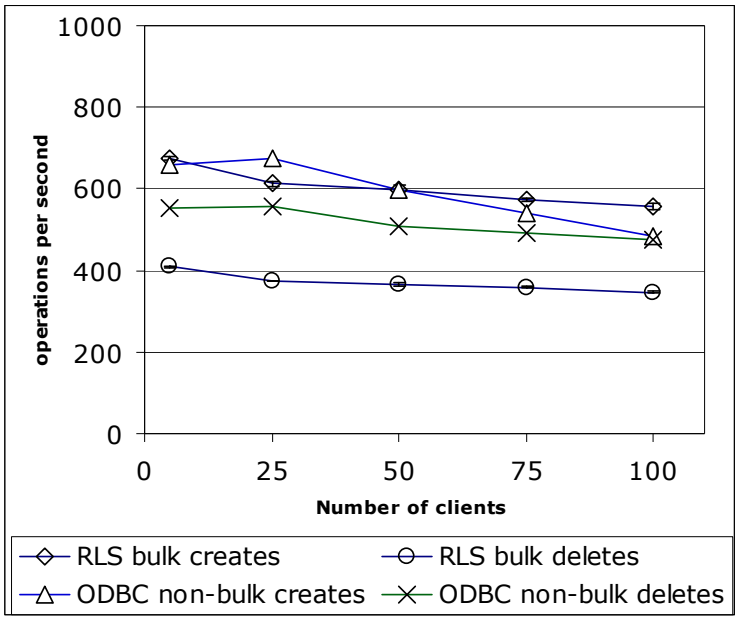


Figure 6: RLS bulk create and delete rates compared to native MySQL rates using an ODBC client.

B. Replica Location Index Query Performance

Figure 7(a) shows RLI bulk query performance using uncompressed LRC updates that are stored in a MySQL database back end. These

RLI query rates are similar to those achieved by the LRC, about 2000 queries per second. Figure 7(b) shows that an RLI that accepts Bloom filter updates and stores them in memory achieves higher query rates (e.g., for 100 clients and one Bloom filter, the RLI supports a factor of five increase over uncompressed updates).

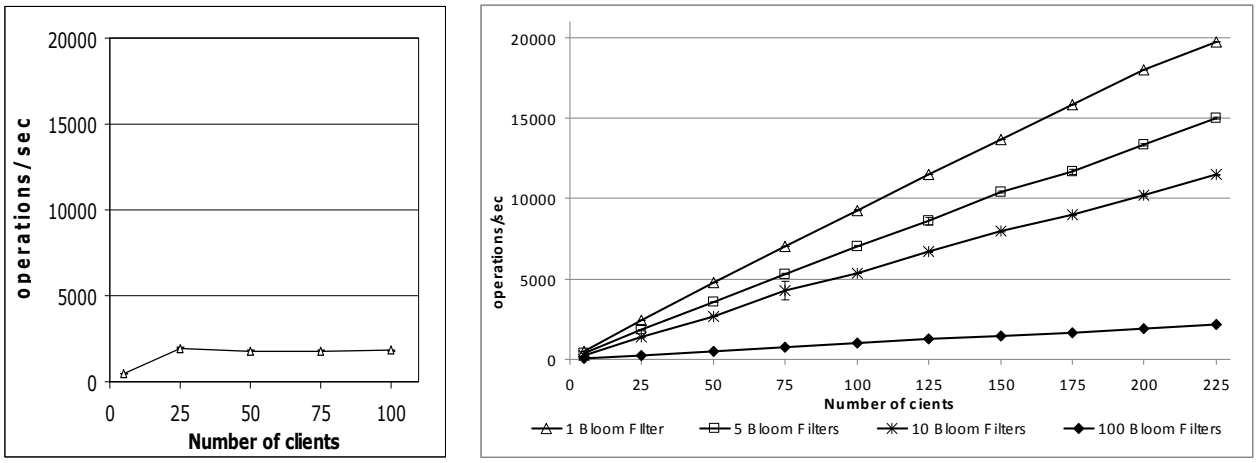


Figure 7: (a) RLI Query Rates With MySQL back end, (b) With in–memory Bloom filter queries.

Figure 8 shows that the time to send uncompressed soft state updates increases with the number of LRCs concurrently updating an RLI, where each LRC in the experiment contains 1 million mappings. Once the maximum update rate for the RLI database back end is reached, this database becomes a performance bottleneck.

These results indicate that performing frequent uncompressed soft state updates does not scale well. We recommend the use of either the incremental mode with uncompressed updates or compression to achieve acceptable scalability. The choice of update mode may depend on whether applications can occasionally tolerate long full updates and whether they require wildcard searches on RLI contents, which are not supported by Bloom filters.

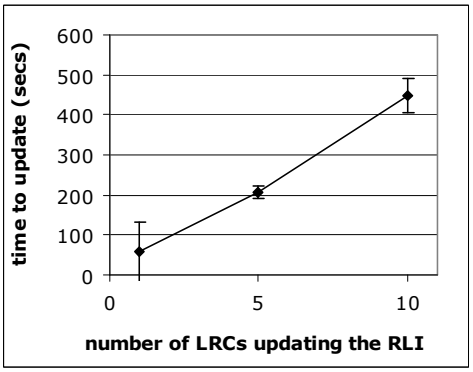


Figure 8: Time taken to send un-compressed updates from LRC to RLI

C. *Soft State Updates Using Bloom Filter Compression*

Next, we present the performance of wide area soft state updates using Bloom filter compression. Updates were sent from LRCs in Los Angeles to an RLI in Chicago (average round trip time 63.8ms). Table 3 shows Bloom filter update statistics for a single client performing a soft state update for a range of LRC database sizes.

TABLE 3
COMPRESSED SOFT STATE PERFORMANCE

Database size (number of mappings)	Average time to perform an update (seconds)	Average time to generate bloom filter (seconds)
1 million	less than 1	8.8
5 million	5.4	45.8
10 million	10.4	98.2

The second column shows that Bloom filter wide area updates are significantly faster then uncompressed updates. For example, a Bloom filter update for an LRC with 1 million entries took less then 1 second in the wide area compared to 58 seconds for an uncompressed update in the LAN (Figure 8). The third column shows the time required compute a Bloom filter for a specified LRC database size. This is a one-time cost, since subsequent updates to LRC mappings can be reflected by setting or clearing the corresponding bits in the Bloom filter.

D. *False Positive Error Rates when Using Bloom Filter Compression*

Bloom filters are compact data structures for compressed representation of set membership. Key design tradeoffs are computational overhead (the number of hash functions used), space overhead (the size of the filter), and the false positive (collision) rate. Fixing any two of these determines the value of the third. Generally, computational overhead and the false positive rate are determined by application requirements, and they dictate the size of the filter. Fan et al. [18] present a theoretical framework for filter design based on the assumption of perfect hashing functions. We have verified that, in practical settings, the behavior of MD5 and SHA2 hashing functions is close to ideal hashing functions and that the Fan et al. model accurately predicts the

rate of false positives. Over a set of 50 experiments with different filter sizes, number of hash functions, and target rates of false positives, the average achieved rate of false positives was within 14% of the predicted rate. More on Bloom filter configuration tradeoffs is in [19].

E. RLI False Negative Responses Using Bloom Filter Compression

Next, we examine the probability that a query to an RLI results in a "false negative" response, meaning that the query result incorrectly indicates that none of the LRCs that update the RLI contains a mapping for the logical name. False negatives occur for three reasons: an update message to the RLI has not been generated yet by the LRC; the update message has been generated but it has not yet propagated; or the update message has been dropped due to an error in the communication layer.

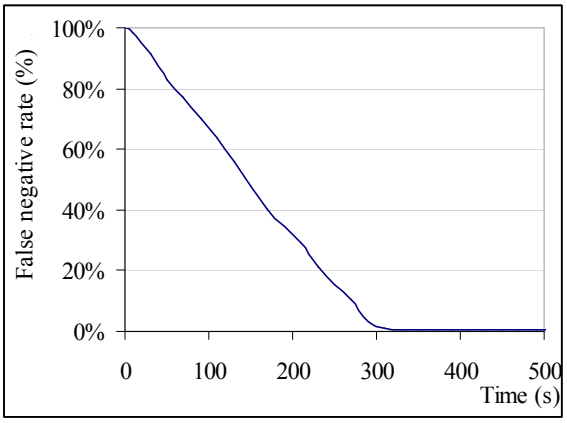


Figure 9: Variation of average false negative rate with time. At time 0 a new LFN to PFM mapping is inserted. The plot shows that after 300s the measured false negative levels down to zero.

To study the probability of false negatives, we deployed a simplified, prototype version of the RLS on 50 PlanetLab hosts, with one LRC and one RLI at each host. At the beginning of the experiment, each LRC stores one million mappings. We play a synthetic trace that simulates client load: the trace generates 10 add operations per second at each participating LRC. Nodes generate an incremental update every time at least 0.5% of their content changes or every five minutes, whichever comes first. Additionally, nodes generate a full update every 100 incremental updates or every 3 hours, whichever comes first. Soft-state timeouts are set at 15 minutes. To understand how fast updates propagate, each node generates queries at the RLI for keys that have been inserted at an LRC at random times after insertion.

Figure 9 presents the average false negatives rate as a function of the time after inserting a key

in the system. The rate of false negatives decreases approximately linearly with time from 100% false negatives immediately after an insertion (since no updates have been generated) to 0.023% at 300 seconds, the period after which all soft-state updates have been generated. This small non-zero false negative rate is a result of one incremental update that is not delivered to its destination by the multicast layer. This is corrected 2070 seconds later by a full update.

VII. The Replica Location Service In Production Use

This section presents several Grid projects that use the RLS in production deployments, often in conjunction with higher-level data management services.

A. LIGO

The Laser Interferometer Gravitational Wave Observatory (LIGO) collaboration [3, 15] conducts research to detect gravitational waves. LIGO uses data replication extensively to make terabytes of data available at ten LIGO sites. The LIGO deployment registers RLS mappings from more than 25 million logical file names to 120 million physical files. The LIGO deployment is a fully-connected RLS: each RLI collects state updates from LRCs at all ten sites. Thus, a query to any RLI identifies all LRCs that contain mappings for a logical file name.

To meet LIGO data publication, replication and access requirements, researchers developed the Lightweight Data Replicator system [8], which integrates RLS, the GridFTP data transfer service [20] and a distributed metadata service. LDR provides rich functionality, including pull-based file replication; efficient data transfer among sites; and a validation component that verifies that files on storage systems are correctly registered in each LRC. Replication of published data sets occurs when a scheduling daemon at each site performs metadata queries to identify *collections* of files with specified metadata attributes and checks the LRC to determine whether the files in a

collection exist on the local storage system; if not, it adds the files to a priority-based transfer queue. A transfer daemon at each site periodically checks this queue, queries the RLI server to find locations where desired files exist, and initiates GridFTP transfer operations to the local site. After the file transfers complete, the transfer daemon registers new files in the local LRC.

B. Earth System Grid

The Earth System Grid (ESG) [5] provides infrastructure for climate researchers that integrates computing and storage resources at five institutions. This infrastructure includes RLS servers at five sites in a fully-connected configuration that contain mappings for over one million files. ESG, like many scientific applications, coordinates data access through a web-based portal. This portal provides an interface that allows users to request specific files or query ESG data holdings with specified metadata attributes. After a user submits a query, the portal coordinates Grid middleware to deliver the requested data. This middleware includes RLS, a centralized metadata catalog, and services for data transfer [20] and subsetting [5]. In addition to using RLS to locate replicas, the ESG web portal uses size attributes stored in RLS to estimate transfer times.

C. RLS and Workflow Management Systems

The Pegasus (Planning for Execution in Grids) workflow mapping system [21] is used by scientific applications to manage complex executions. Components of workflows are often scientific applications and data movement operations; dependencies among these components are reflected in the workflow. Pegasus maps from a high-level, abstract definition of a workflow to a concrete or executable workflow in the form of a Directed Acyclic Graph (DAG), which is then passed to the Condor DAGMan execution system [22]. During workflow mapping, Pegasus queries the RLS to identify and select physical replicas specified as logical files in the abstract

workflow. When Pegasus produces new data files during execution, it registers them in the RLS.

Pegasus is used for production analysis by a number of scientific applications, including the LIGO [3] project, the Montage mosaic [23] and the Galaxy morphology [24] astronomical applications, and the Southern California Earthquake Center [7] project.

VIII. Ongoing and Future Work

In addition to ongoing improvements in the performance, scalability and robustness of RLS, we are investigating peer-to-peer organizations and higher-level data services that integrate RLS.

A. Peer-to-Peer Replica Location Services

We have studied two approaches to peer-to-peer organization of RLS servers to support self-organization, greater fault-tolerance and improved scalability.

First, we designed and implemented a Peer-to-Peer Replica Location Service (P-RLS) [12] that uses a *structured overlay* network (Chord [25]) to organize participating 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 server responds to queries for the mappings it contains and routes other queries to the P-RLI nodes that contain the corresponding mappings. Our prototype extends the Globus Version 3.0 RLS server with Chord protocols.

Second, we implemented and evaluated a decentralized, adaptive replica location mechanism [13] based on an *unstructured overlay,* Bloom filter compression and soft state updates. This scheme uses the unstructured overlay to distribute 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 an LFN mapping, the node first checks its locally stored mappings and its stored digests

that summarize the contents of remote nodes. If the node finds a match for the LFN in a digest for a remote node, it contacts that node to obtain the mappings. Network traffic generated using this approach is comparable to the traffic generated in query forwarding schemes. Query performance improves significantly because all queries are resolved in at most two network hops.

B. Higher Level Data Management: The Data Replication Service

As demonstrated by the systems described in Section VII, there is a need for higher-level Grid data management capabilities that include replica registration and discovery. We are designing several higher-level data management services. One example is the Data Replication Service (DRS) [26], which ensures that a specified set of files is replicated on a storage site. DRS operations include *location*, identifying where desired data files exist on the Grid by querying the RLS; *transfer*, moving desired data files to the local storage system efficiently; and *registration*, adding location mappings to the RLS so that clients may discover newly-created replicas.

IX. RELATED WORK

Several areas of related work are relevant to the RLS: other replica location and management systems in Grid environments; solutions for locating resources in peer-to-peer networks; distributed name systems; distributed file systems and database catalogs; and web proxy caches.

A. Grid Replica Location and Management

Other Grid approaches to replica management include the Storage Resource Broker [27] and GridFarm [28] systems that manage replicas using a centralized metadata catalog. Unlike RLS, these catalogs maintain logical metadata to describe the content of data in addition to attributes related to physical replicas; these catalogs are also used to maintain replication consistency.

The European DataGrid project implemented a Replica Location Service based on our jointly

developed RLS framework [2] as part of a replica management architecture [29] for data produced by the Large Hadron Collider (LHC) at CERN.

The LHC Computing Grid project (LCG) and the Enabling Grids for E-SciencE (EGEE) project at CERN have implemented *file catalogs* that combine replica location functionality and hierarchical file system metadata management. The LCG catalog is a connection-oriented, stateful server [30, 31]. The EGEE File and Replica Management (FiReMan) catalog uses a service-oriented architecture and supports bulk operations [32].

B. Peer-to-Peer Resource Discovery

Resource discovery is an important problem for peer-to-peer (P2P) networks. Approaches include centralized indexing, flooding-based *unstructured* schemes, and Distributed Hash Tables (DHTs) and other *structured* approaches.

Systems like Napster maintain a centralized index server, and each resource discovery operation queries this server. This approach has limited scalability and a single point of failure.

Gnutella [33, 34] constructs an *unstructured* overlay network among nodes and uses flooding to distribute queries. Although query flooding improves fault tolerance by eliminating the single point of failure, it may introduce large message traffic and processing overheads in the network [35-37] and does not scale well. These overheads are reduced by bounding the number of times each message is forwarded by the time-to-live field of query messages or by employing one of several random walk and replication schemes (e.g., Gia [34, 38]).

Distributed hash table (DHT) approaches construct a structured overlay and utilize message routing instead of flooding to discover a resource [39]. DHT systems (e.g., Chord [25], CAN [40], Tapestry [41]) perform file location by hashing logical identifiers into keys. Each node maintains location information for a subset of the hashed keys and supports searches by passing

queries according to the overlay structure. Some systems (e.g. OceanStore [42]) also employ probabilistic search to reduce search latencies.

As described in Section VIII.A, we have experimented with two P2P RLS implementations, one based on a structured DHT system [12] and one based on an unstructured P2P overlay [13].

C. Distributed Name Services

The functionality provided by RLS has some similarities to that provided by distributed name services such as the Domain Name System and the Handle System.

The Domain Name System (DNS) [43, 44] is a highly scalable, hierarchical, globally distributed database that maps human-readable hostnames to IP addresses. Root servers at the top of the namespace hierarchy are centrally managed and replicated at multiple locations for high availability. Subdomains of the hierarchical namespace are administered locally. RLS has similarities with DNS in its hierarchical organization and local administration of name mappings in LRCs. However, RLS does not does not provide global name mappings, enforce namespaces or restrict the targets of name mappings (i.e., they need not be IP addresses).

The Handle System (HS) [45, 46] is another scalable, efficient, distributed global name service. Its organization is hierarchical, with a top-level Global Handle Registry and an unlimited number of lower-level Local Handle Services that manage local namespaces. A handle is a globally-unique identifier that may "refer to multiple instances of a resource" [45], providing one-to-many mapping functionality similar to that in RLS. The HS differs from DNS in its support for general-purpose handle to attribute mappings rather than mappings from domain names to IP addresses for network routing, its administrative model, and its security features. The HS supports many features similar to RLS functionality, with an additional focus on global namespace management. We may provide an RLS implementation based on Handle in the future.

D. Distributed Databases

Heterogeneous distributed databases use catalogs to keep track of information that is needed to locate and manage distributed objects [47-49] such as relations, views and indexes as well as information about users, query plans and access privileges [47]. Like RLS, these systems typically include local catalogs that contain information needed to manage a local Database Management System and a distributed, inter-site catalog for managing distributed resources. The R* system [48] has a partitioned, distributed catalog architecture, where each site maintains a catalog with information about a set of objects for which it is responsible. Each object's name includes a "birth site" for the object that never changes. The current storage location of an object is obtained from its birth site.

Choy and Selinger extend the R* catalog [49]. Their architecture includes a Registration Site that can be inferred from an object's name and never changes. This site maps an object's name to a Catalog Site that in turn contains a mapping to the Storage Site where the object is currently stored. This additional indirection provides "flexibility in the placement of catalog information and allows a transparent relocation of catalog information" [49]. This scheme supports a variety of catalog configurations, including centralized, fully replicated and partitioned catalogs.

The Group Oriented Catalog Allocation Scheme [47] is a hybrid catalog architecture between a strictly partitioned catalog scheme like R* and a fully replicated distributed catalog. It partitions information among groups of catalogs and fully replicates information within a group. This scheme reduces update costs compared to a fully replicated catalog and reduces query times compared to a partitioned catalog.

E. Internet Proxy Caches

Directory services for cooperative Internet proxy caches [18, 50-52] offer functionality similar to the RLS. In a cooperative proxy cache, the directory service receives requests for a URL and locates one proxy that caches a replica of that URL. The RLS provides additional functionality, including requests for multiple replicas of the same file and batch requests.

Hierarchical Internet proxy-caches have been extensively analyzed [52]. Three additional solutions do not use hierarchies. In the Summary Cache [18], each pair of participating proxies periodically exchanges Bloom filter summaries of the data stored locally. The RLS Bloom filter scheme, by contrast, uses an adaptive soft-state protocol to distribute digests. The Cache Array Routing Protocol (CARP) [53] uses consistent hashing [54, 55] to partition the name space and route requests in a manner similar to a DHT scheme. The CARP solution, however, does not efficiently support queries for co-located sets of files. In the Vicinity Cache [56], a directory service at each node considers data exchange costs (e.g. latency or bandwidth) when disseminating information to other nodes.

X. CONCLUSION

We have described the design, implementation, evaluation and deployment of the Globus Replica Location Service. Our design is based on a general framework that allows a distributed RLS to be configured based on required system scale and reliability. We have described the RLS implementation and presented a performance study that demonstrates that individual RLS servers perform well and scale up to millions of entries and one hundred clients and that soft state updates of the distributed index scale well using Bloom filter compression. Today, RLS is used in a variety of production Grid environments. The largest of these deploys RLS at ten sites and

manages over 25 million unique logical files and 120 million physical replicas. RLS has been successfully integrated into a variety of customized, higher-level data management systems.

ACKNOWLEDGMENT

Work Package 2 of the DataGrid Project co-designed the RLS framework and evaluated the performance of early RLS versions. Keith Jackson implemented the Python RLS client. We appreciate the efforts of Scott Koranda and the LIGO collaboration, Luca Cinquini and the ESG project, and Gaurang Mehta, Ewa Deelman and the Pegasus group in deploying and testing RLS.

REFERENCES

[1] A. L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, R. Schwartzkopf, "Performance and Scalability of a Replica Location Service," presented at Thirteenth IEEE Int'l Symposium High Performance Distributed Computing (HPDC-13), Honolulu, HI, 2004.

[2] 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, 2002.

[3] A. Abramovici, W. Althouse, R. Drever, Y. Gürsel, S. Kawamura, F. Raab, D. Shoemaker, L. Sievers, R. Spero, K. Thorne, R. Vogt, R. Weiss, S. Whitcomb, and M. Zucker, "LIGO: The Laser Interferometer Gravitational-Wave Observatory," *Science*, vol. 256, pp. 325-333, 1992.

[4] ESG Project, "The Earth System Grid, www.earthsystemgrid.org," 2005.

[5] 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, D. Williams, "The Earth System Grid: Supporting the Next Generation of Climate Modeling Research," *Proceedings of the IEEE*, vol. 93, pp. 485- 495, 2005.

[6] QCDGrid Project, "QCDGrid: Probing the Building Blocks of Matter with the Power of the Grid, http://www.gridpp.ac.uk/qcdgrid/," 2005.

[7] SCEC Project, "Southern California Earthquake Center, http://www.scec.org/," 2005.

[8] LIGO Project, "Lightweight Data Replicator, http://www.lsc-group.phys.uwm.edu/LDR/," 2004.

[9] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, and R. Cavanaugh, "Mapping Abstract Complex Workflows onto Grid Environments," *Journal of Grid Computing*, vol. 1, pp. 25-39, 2003.

[10] Safehaus, "Java UUID Generator, http://jug.safehaus.org/," 2008.

[11] B. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of ACM*, vol. 13, pp. 422-426, 1970.

[12] M. Cai, A. Chervenak, M. Frank, "A Peer-to-Peer Replica Location Service Based on a Distributed Hash Table," presented at SC2004 Conference, Pittsburgh, PA, 2004.

[13] 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, 2002.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," presented at Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles (SOSP), 2007.

[15] LIGO Project, "LIGO - Laser Interferometer Gravitational Wave Observatory, http://www.ligo.caltech.edu/," 2004.

[16] L. Lamport, "Concurrent reading and writing," *Communications of the ACM*, vol. 20, pp. 806-811, 1977.

[17] K. Czajkowski, et al., "The WS-Resource Framework Version 1.0, http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf," 2004.

[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, vol. 8, pp. 281-293, 2000.

[19] M. Ripeanu, "Using peer-to-peer experience to build large scale Grid services, PhD Dissertation," vol. PhD The University of Chicago, 2005.

[20] Globus Project, "The GridFTP Protocol and Software," 2002.

[21] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, M. Livny, "Pegasus : Mapping Scientific Workflows onto the Grid," presented at Across Grids Conference, Nicosia, Cyprus, 2004.

[22] J. Frey, T. Tannenbaum, I. Foster, M. Livny, S. Tuecke, "Condor-G: A computation management agent for multiinstitutional grids," *Cluster Computing*, vol. 5, pp. 237-246, 2002.

[23] G. B. Berriman, et al. , "Montage: A Grid-Enabled Image Mosaic Service for the NVO," presented at Astronomical Data Analysis Software & Systems (ADASS) XIII, 2003.

[24] E. Deelman, et al., "Grid-Based Galaxy Morphology Analysis for the National Virtual Observatory," presented at SC2003, 2003.

[25] 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.

[26] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, B. Moe, "Wide Area Data Replication for Scientific Collaborations," presented at 6th IEEE/ACM Int'l Workshop on Grid Computing (Grid2005), Seattle, WA, USA, 2005.

[27] A. Rajasekar, et al., "Storage Resource Broker - Managing Distributed Data in a Grid," *Computer Society of India Journal, Special Issue on SAN*, vol. 33, pp. 42-54, 2003.

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

[29] P. Kunszt, Erwin Laure, Heinz Stockinger, Kurt Stockinger, "Advanced Replica Management with Reptor," presented at 5th International Conference on Parallel Processing and Applied Mathematics, Czestochowa, Poland, 2003.

[30] J. P. Baud, J. Casey, S. Lemaitre, and C. Nicholson, "Performance analysis of a file catalog for the LHC computing grid," presented at 14th IEEE International Symposium on High Performance Distributed Computing(HPDC-14), 2005.

[31] C. Munro and B. Koblitz, "Performance Comparison of the LCG2 and gLite File Catalogues," *Nuclear Instruments and Methods in Physics Research Section A*, vol. 559, pp. 48-52, 2006.

[32] P. F. Kunszt, P. Badino, A. Frohner, G. McCance, K. Nienartowicz, R. Rocha, and D. Rodrigues, "Data Storage, Access and Catalogs in gLite," presented at Local to Global Data Interoperability-Challenges and Technologies, 2005.

[33] M. Ripeanu, "Peer-to-Peer Architecture Case Study: Gnutella Network," presented at IEEE 1st International Conference on Peer-to-peer Computing (P2P2001), Linkoping, Sweden, 2001.

[34] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making Gnutella-like P2P Systems Scalable," presented at ACM SIGCOMM 2003, Karlsruhe, Germany, 2003.

[35] 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, pp. 50-57, 2002.

[36] 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.

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

[38] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and Replication in Unstructured Peer-to-Peer Networks," presented at 16th ACM International Conference on Supercomputing(ICS'02), New York, USA, 2002.

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

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

[41] B. Y. Zhao, et al., "Tapestry: A Resilient Global-scale Overlay for Service Deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, 2004.

[42] J. Kubiatowicz, et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," presented at 9th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), 2000.

[43] P. Mockapetris and K. J. Dunlap, "Development of the domain name system," presented at ACM Symposium on Communications Architectures and Protocols (SIGCOMM '88), Stanford, CA, USA, 1988.

[44] J. Jung, E. Sit, H. Balakrishnan, and R. Morris, "DNS performance and the effectiveness of caching," *IEEE/ACM Transactions on Networking (TON)*, vol. 10, pp. 589-603, 2002.

[45] S. Sun, L. Lannom, and B. Boesch, "Handle System Overview. Internet Engineering Task Force (IETF) Request for Comments (RFC)," RFC 3650, November 2003. http://hdl. handle. net/4263537/4069.

[46] R. Kahn and R. Wilensky, "A framework for distributed digital object services," *International Journal on Digital Libraries*, vol. 6, pp. 115-123, 2006.

[47] H. Cho, "Catalog management in heterogeneous distributed database systems," presented at Proceedings of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 1997.

[48] B. G. Lindsay, "A retrospective of R: A distributed database management system," *Proceedings of the IEEE*, vol. 75, pp. 668-673, 1987.

[49] D. M. Choy, P. G. Selinger, I. Center, and C. A. San Jose, "A distributed catalog for heterogeneous distributed database resources," presented at Proceedings of the First International Conference on Parallel and Distributed Information Systems, 1991.

[50] Squid-cache Project, " Squid: Optimising Web Delivery, http://www.squid-cache.org/," 2008.

[51] M. Beynon, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman and Joel Saltz, "Distributed Processing of Very Large Datasets with DataCutter," *Parallel Computing*, vol. 27, pp. 1457-1478, 2001.

[52] 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, 2003.

[53] J. Liang, R. Kumar, and K. W. Ross, "The KaZaA Overlay: A Measurement Study," *Computer Networks Journal (Elsevier)*, vol. 49, 2005.

[54] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, "Web Caching with Consistent Hashing," presented at The Eighth International World Wide Web Conference (WWW8), Toronto, Canada, 1999.

[55] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," presented at Symposium on Theory of Computing, 1997.

[56] M. Rabinovich, J. Chase, and S. Gadde, "Not All Hits Are Created Equal: Cooperative Proxy Caching Over a Wide-Area Network," presented at Third International WWW Caching Workshop, 1998.

Ann L. Chervenak is a Project Leader at Information Sciences Institute of the University of Southern California and a member of the research faculty of the USC Computer Science Department. She leads research efforts in the area of distributed data management for scientific applications, including climate modeling, gravitational wave physics, health grids and others. She received her M.S. and Ph.D. in Computer Science from UC Berkeley.

Robert Schuler received the BS degree in Computer Science from USC in 1995. He is currently a senior programmer/analyst at USC Information Sciences Institute. Earlier, he held senior engineering positions in an emerging technologies group of Xerox Corporation and a technology startup serving leading media companies and consortiums. His research interests include distributed computing, grid computing and scientific data management.

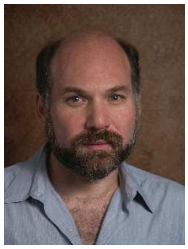
Matei Ripeanu received his Ph.D. degree in Computer Science from The University of Chicago in 2005. After a brief visiting period with Argonne National Laboratory, Matei joined the Electrical and Computer Engineering Department of the University of British Columbia as an Assistant Professor. Matei is broadly interested in distributed systems with a focus on self-organization and decentralized control in large-scale Grid and peer-to-peer systems.

Muhammad Ali Amer is a Ph.D. candidate in Computer Science Department at the University of Southern California. He received his B.S. degree from National University of Sciences and Technology, Pakistan, in 2001 and his M.S. in Computer Science from USC in 2006.

Shishir Bharathi is a Ph.D. candidate in the USC Computer Science Department. He earned a B.E. in Computer Science and Engineering from Mysore University, India and an M.S. in CS from USC. His current research focuses on efficient data management for the execution of large scale scientific workflows on the Grid. He has also worked on the Globus Replica Location Service and on integrating peer-to-peer methodologies with Grid information services.

Ian Foster is an Argonne Distinguished Fellow, the Arthur Holly Compton Distinguished Service Professor of Computer Science, and Director of the Computation Institute at Argonne National Laboratory and the University of Chicago. His research interests are in distributed and parallel computing and computational science, and he has published six books and over 200 articles and technical reports on these and related topics. His Distributed Systems Laboratory is a leading contributor to the Globus Toolkit, widely used open source Grid software, and also plays a leading role in projects applying Grid technologies to scientific and engineering problems in such fields as high energy physics, climate data analysis, and cancer biology.

Adriana Iamnitchi is Assistant Professor in the Computer Science and Engineering Department of University of South Florida. Adriana received her Ph.D. in Computer Science from the University of Chicago in 2003. Her research interests are in distributed systems with a focus on self-organization and decentralized control in large-scale Grid and peer-to-peer systems.



Carl Kesselman is a Professor in the Viterbi School of Engineering at USC with a appointments in Systems Engineering and Computer Science. He is also co-director of the Center for Health Informatics in the USC Information Sciences Institute. He received his Ph.D. in CS from the University of California at Los Angeles. Dr. Kesselman's research interests focus on how globally distributed shared infrastructure can be used to facilitate the creation of distributed, multi-institutional collaborations, or *virtual organizations*.