Performance and Scalability of a Replica Location Service

Ann L. Chervenak, Naveen Palavalli, Shishir Bharathi, Carl Kesselman, Robert Schwartzkopf
University of Southern California Information Sciences Institute
{annc, palavall, shishir, carl, bobs}@isi.edu

Abstract

We describe the implementation and evaluate the performance of a Replica Location Service that is part of the Globus Toolkit Version 3.0. A Replica Location Service (RLS) provides a mechanism for registering the existence of replicas and discovering them. Features of our implementation include the use of soft state update protocols to populate a distributed index and optional Bloom filter compression to reduce the size of these updates. Our results demonstrate that RLS performance scales well for individual servers with millions of entries and up to 100 requesting threads. We also show that the distributed RLS index scales well when using Bloom filter compression for wide area updates.

1. Introduction

Managing replicated data in Grid environments is a challenging problem. Data-intensive applications may produce data sets on the order of terabytes or petabytes. These data sets may be replicated within the Grid environment for reliability and performance. Clients require the ability to discover existing data replicas and create and register new replicas.

A Replica Location Service (RLS) is one component of a Grid data management architecture. An RLS provides a mechanism for registering the existence of replicas and discovering them. In an earlier paper [1], we described a flexible RLS framework that allows the construction of a variety of replica location services with different performance, reliability and overhead characteristics. The RLS framework was co-developed by the Globus and DataGrid projects.

In this paper, we describe a Replica Location Service implementation based on our earlier framework. We evaluate the performance and scalability of individual RLS servers and the overall distributed system.

In addition to Replica Location Services, other components of a Grid replica management system may include consistency services, selection services that choose replicas based on the current state of Grid resources, and data transport protocols and services. These components are outside the scope of this paper.

2. The RLS Framework

The RLS framework [1] upon which our implementation is based has five elements:

- *Local Replica Catalogs (LRCs) that contain mappings from logical to target names*
- *Replica Location Indexes (RLIs) that aggregate state information about one or more LRCs with relaxed consistency*
- *Soft state update mechanisms used to maintain RLI state*
- *Optional compression of soft state updates*
- *Management of RLS member services*

Local Replica Catalogs (LRCs) maintain mappings between logical names and target names. Logical names are unique identifiers for data content that may have one or more physical replicas. Target names are typically the physical locations of data replicas, but they may also be other logical names representing the data. Clients query LRC mappings to discover replicas associated with a logical name.

In addition to local catalogs, we also provide a distributed higher-level *Replica Location Index*. Each RLI server aggregates and answers queries about mappings held in one or more LRCs. An RLI server contains a set of mappings from logical names to LRCs. A variety of index structures can be constructed with different performance and reliability characteristics by varying the number of RLIs and the amount of redundancy and partitioning among them. Figure 1 shows one example configuration.

Information in the distributed RLIs is maintained using *soft state update protocols*. Each LRC sends information about its mappings to zero or more

RLIs. Information in RLIs times out and must be periodically refreshed by subsequent soft state updates. An advantage of using soft state update protocols is that we are not required to maintain persistent state for an RLI. If an RLI fails and later resumes operation, its state can be reconstructed using soft state updates.

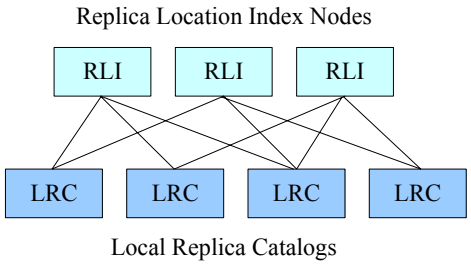

Figure 1: Example Replica Location Service configuration

Soft state updates may optionally be *compressed* to reduce the amount of data sent from LRCs to RLIs and reduce storage and I/O requirements on RLIs. The RLS framework paper proposed several compression options, including compression based on logical collections and the use of Bloom Filter compression [2][3], in which bit maps are constructed by applying a series of hash functions to logical names. The framework paper also proposed partitioning of LRC updates based on the logical name space to reduce the size of soft state updates.

The final component of the RLS framework is a *membership service* that manages the LRCs and RLIs participating in a Replica Location Service and responds to changes in membership, for example, when a server enters or leaves the RLS. Membership changes may result in changes to the update patterns among LRCs and RLIs.

3. An RLS Implementation

Based on the RLS framework above, we have implemented a Replica Location Service that is included in the Globus Toolkit Version 3.0. In this section, we describe features and design choices made for our implementation.

3.1 The Common LRC and RLI Server

Although the RLS framework treats the LRC and RLI servers as separate components, our implementation consists of a common server that can be configured as an LRC, an RLI or both. Figure 2 shows the server design.

The RLS server is multi-threaded and is written in C. The server supports Grid Security Infrastructure (GSI) authentication. An RLS server may have an associated gridmap file that maps from Distinguished Names (DNs) in users' X.509 certificates to local usernames. Authorization to perform particular RLS operations is granted to users based on access control lists that are included in the server configuration. Access control list entries are regular expressions that grant privileges such as lrc_read and lrc_write access to users based on either the Distinguished Name (DN) in the user's X.509 certificate or based on the local username specified by the gridmap file. The RLS server can also be run without any authentication or authorization, allowing all users the ability to read and write RLS mappings.

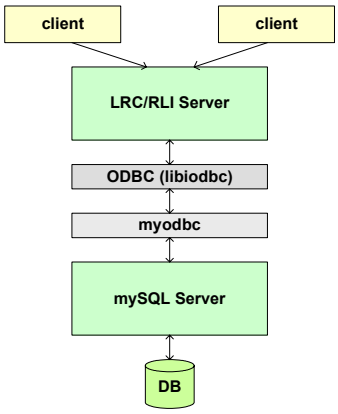

Figure 2: RLS Implementation

The RLS server back end is a relational database. Because we use an Open Database Connectivity (ODBC) layer between the server and the relational database back end, it is relatively easy to provide support for a variety of relational database back ends. Currently supported back ends include MySQL and PostgreSQL. RLS versions 2.1.3 and later also support an Oracle database back end.

The table structure of the LRC relational database back end is relatively simple and is shown in Figure 3. It contains 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). Typically these attributes are used to associate such values as size with a physical name for a file or data object. Finally, there is a table that lists all RLIs updated by the LRC and one that stores regular expressions for LRC namespace partitioning.

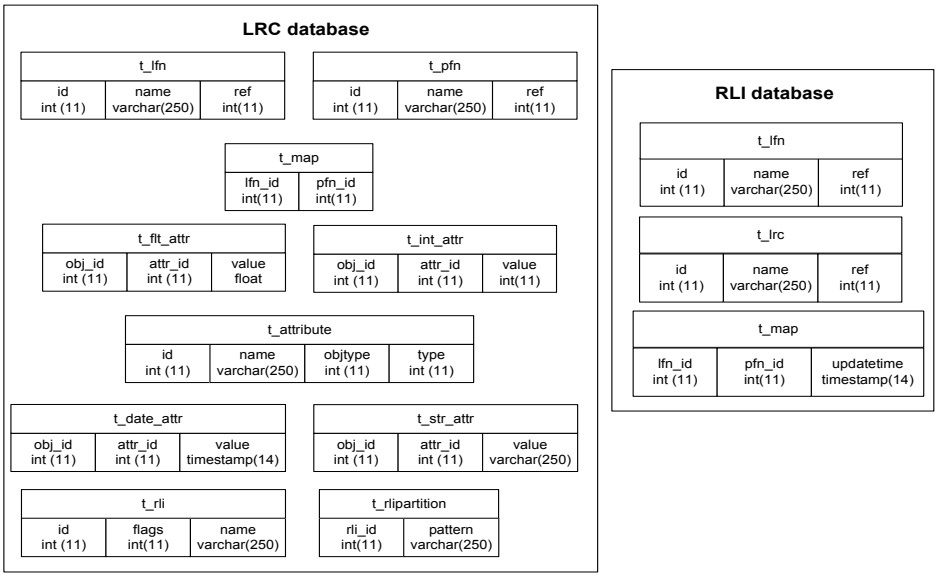

Figure 3: Relational tables used in LRC and RLI database implementations

Typically, an external service populates the LRC to reflect the contents of a local file or storage system. Alternatively, a workflow manager or a data publishing service that generates new data items may register them with the RLS.

In version 2.0.9 of the RLS, which is evaluated in this paper, the RLI server uses a relational database back end when it receives full, uncompressed updates from LRCs. This relational database contains three simple tables, as shown in Figure 3: one for logical names, one for LRCs and a mapping table that stores {LN, LRC} associations. When an RLI receives soft state updates using Bloom filter compression (described below), no database is used in the RLI; Bloom filters are instead stored in RLI memory.

3.2 Soft State Updates

Local Replica Catalogs send periodic summaries of their state to Replica Location Index servers. In our RLS implementation, soft state updates may be of four types: uncompressed updates, those that combine infrequent full updates with more frequent incremental updates, updates using Bloom filter compression [2], and updates using name space partitioning.

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 LRCs. 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 logical name. Then the client queries LRCs to obtain the target name mappings.

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

When using soft state updates, there is some delay between when changes are made in LRC mappings and when those changes are reflected in RLIs. Thus, a query to an RLI may return stale information. In this case, a client may not find a mapping for the desired logical name when it queries an LRC. An application program must be sufficiently robust to recover from this situation and query for another replica of the logical name.

3.3 Immediate Mode

To reduce both the frequency of full soft state updates and the staleness of information in an RLI, our implementation supports an incremental or *immediate mode* where infrequent full updates are combined with more frequent incremental updates that reflect recent changes to an LRC. Immediate mode updates are sent after a short, configurable interval has elapsed (by default, 30 seconds) or after a specified number of LRC updates have occurred. Periodic full updates are required because RLI information eventually expires and must be refreshed. In practice, the use of immediate mode is almost always advantageous. The only exception is when large numbers of mappings are loaded into an LRC server at once, for example, during initialization of a new server.

3.4 Compression

The compression scheme provided by our implementation uses Bloom filters, which are arrays of bits [2]. A Bloom filter that summarizes the state of an LRC is constructed by performing multiple hash functions on each logical name registered in the LRC and setting the corresponding bits in the Bloom filter. The resulting bit map is sent to an RLI, which stores one Bloom filter per LRC. For RLS version 2.0.9, no relational database back end is deployed for RLIs that receive Bloom filter updates. Rather, all Bloom filters are stored in memory, which provides fast soft state update and query performance.

When an RLI receives a query for a logical name, it performs the same hash functions against the logical name and checks whether the corresponding bits in each LRC Bloom filter are set. If the bits are not set, then the logical name is not registered in the corresponding LRC. However, if the bits are set, there is a small possibility that a false positive has occurred, i.e., a false indication that the LRC contains a mapping for that logical name. The probability of false positives is determined by the size of the Bloom filter bit map as well as the number of hash functions calculated on each logical name. Our implementation sets the Bloom filter size based on the number of mappings in an LRC (e.g., 10 million bits for approximately 1 million entries). We calculate three hash values for every logical name. These parameters give a false positive rate of approximately 1%. Different parameters can produce an arbitrarily small rate of false positives, at the cost of larger bit maps or more overhead for calculating hash functions.

3.5 Partitioning

Finally, our implementation supports partitioning of soft state 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. Partitioning is rarely used in practice because complete Bloom filter updates are efficient to compute and greatly reduce the size of soft state updates.

3.6 Membership service

Our current implementation does not include a membership service that manages LRCs and RLIs participating in the distributed system. Instead, we use a simple static configuration of LRCs and RLIs. As the RLS implementation evolves into a Web service implementation [4][5], we will implement a membership service on top of registries provided by Web service environments

3.7 RLS Clients

The RLS implementation includes two client interfaces, one written in C and one that provides a Java wrapper around the C client. 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 database tables.

Table 1: Summary of LRC and RLI Operations

LRC Operations	
Mapping management	Create mapping, add, delete, bulk create,bulk add, bulk delete
Attribute management	Create attribute, add, modify, delete, bulk create, bulk add, bulk modify, bulk delete
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

4. Methodology for Performance Study

Unless otherwise indicated, the software versions used in our performance study are those indicated in Table 2.

Table 2: Software versions used

Replica Location Service	Version 2.0.9
Globus Packaging Toolkit	Version 2.2.5
libiODBC library	Version 3.0.5
MySQL database	Version 4.0.14
MyODBC library (with MySQL)	Version 3.51.06
PostgreSQL database	Version 7.2.4
Psqlodbc library(with PostgreSQL)	Version 7.3.1

Our first set of tests evaluates the performance of individual Local Replica Catalogs (LRCs) and Replica Location Indexes (RLIs). We submit requests to these

catalogs using a multi-threaded client program written in C that allows the user to specify the number of threads that submit requests to a server and the types of operations to perform (add, delete, or query mappings). We typically initiate 3000 operations for add trials and 20,000 or more operations for query trials to achieve efficient server performance and determine the rate of operations. For each performance number reported in our study, we perform several trials (typically 5) and calculate the mean rate over those trials. For each set of trials, a server is loaded with a predefined number of mappings. The database size is kept relatively constant during a performance test. For example, in case of add tests, the mappings that are added in each trial are deleted before subsequent trials are performed.

The second set of tests measures soft state update performance between LRC and RLI servers. We measure the performance of uncompressed updates as well as updates that use Bloom filter compression. For these tests, LRC servers are loaded with a predefined number of mappings and are forced to update an RLI server. The time taken for soft state updates to complete is measured from the LRC's perspective.

5. Performance and Scalability of the RLS Implementation

In this section, we present performance and scalability results for our RLS implementation. First, we present operation rates for adds, deletes and queries for LRCs with a MySQL relational database back end. Next, we demonstrate the importance of sensitivity to back end characteristics by measuring the effect of garbage collection in the PostgreSQL database. We also present query performance for RLIs that use uncompressed and Bloom filter soft state updates. We demonstrate that uncompressed soft state updates don't scale well for an RLS that contains a large number of replica mappings, suggesting the need to use immediate mode or compression. Finally, we demonstrate good scalability with Bloom filter compression.

5.1 LRC Performance for MySQL Back End

In this set of experiments we present LRC performance results for a MySQL relational database back end. The clients in this test were dual Pentium-III 547 MHz workstations with 1.5 Gigabytes of memory. The server was a dual Intel Xeon 2.2 GHz processor with 1 Gigabyte of memory. The clients and server were on the same 100 megabit per second local area network.

First, we show that LRC operation rates depend on whether the database back end immediately flushes transactions to the physical disk. If the user disables this immediate flush, then transaction updates are instead written to the physical disk periodically. This maintains loose consistency, providing improved performance at some risk of database corruption.

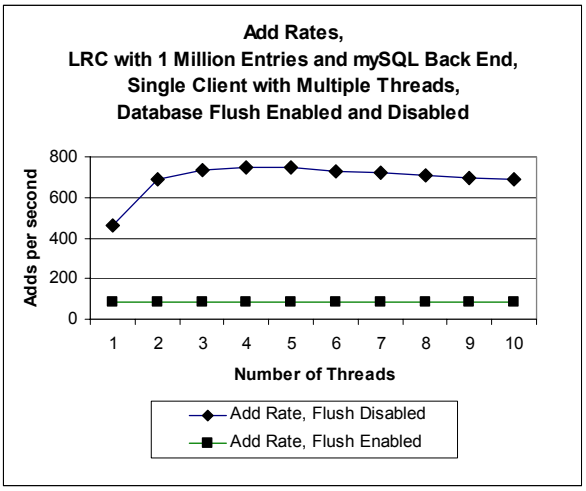

Figure 4: Add Rates for LRC with MySQL back end with flush enabled and disabled.

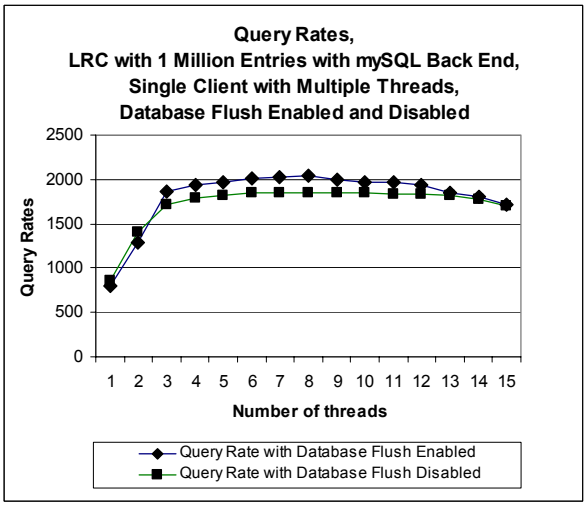

Figure 5: Query Rates for LRC with MySQL back end with flush enabled and disabled.

Figure 4 and Figure 5 show the performance of add and query operations, respectively, for an LRC with a MySQL back end with 1 million entries when the database flush is enabled and disabled. For these tests, the client OS version was Red Hat Linux 9 and the Server OS version was Red Hat Linux 7.2. Operations are issued by a single client with multiple threads. For add operations, there is a significant performance

difference when the database flush is enabled and disabled, with add rates of approximately 84 adds per second and over 700 per second, respectively. By contrast, there is little difference in query performance in Figure 5 whether the database flush is enabled or disabled, since query operations do not change the contents of the database or generate transactions.

Because of the significant performance improvement offered for update operations by disabling the immediate database flush, we recommend that RLS users disable this feature. The remainder of our performance results in this paper will reflect the database flush being disabled, both for the MySQL and the PostgreSQL databases.

Figure 6 shows operation rates when multiple clients with ten threads per client are issuing operations to a single LRC. The same server described above was running the Debian Linux 3.0 operating system during this test. The LRC achieves query rates of 1700 to 2100 per second, add rates of 600 to 900 per second and delete rates of 470 to 570 per second. The rates drop as the total number of threads increases. Query and delete rates drop about 20% and add rates drop about 35% when increasing from 10 to 100 requesting threads.

For comparison, Figure 7 shows native MySQL database performance for similar operations. For this test, we imitated the same SQL operations performed by an LRC for query, add and delete operations but made these requests directly to the MySQL back end. These results show that the LRC adds some overhead compared to the native MySQL database. This overhead is highest for query operations, where the LRC server achieves about 80% of the native MySQL query rate for a single client with 10 threads and about 70% of the native database performance for 10 clients with 100 threads. The overheads are lower for add and delete operations. Add rates on the LRC for a single client are about 89% of the native database performance. Add performance is actually better for the LRC than for the MySQL native database with 10 clients (100 threads). We speculate that managing connections to 100 requesting threads and servicing add requests produces more overhead on MySQL than when requests are submitted through the LRC. The LRC achieves a delete rate of about 87% of the performance of the MySQL database for a single client and about 96% of the native database performance for 10 clients.

We are currently characterizing the source of RLS overheads. We speculate that overhead is incurred in authentication operations, thread management and using globus_IO libraries and our RPC protocol.

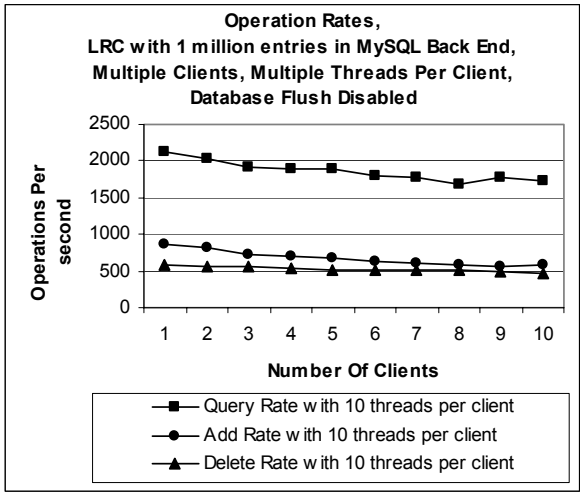

Figure 6: Operation Rates for LRC with MySQL back end.

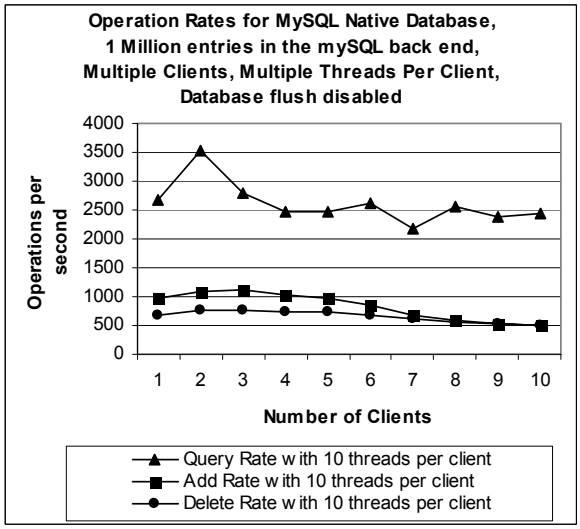

Figure 7: Operation rates for native MySQL relational database performing similar SQL operations to those performed by the LRC.

5.2 LRC Performance with PostgreSQL

Sensitivity to the performance characteristics of the relational database back end is an important issue for those deploying the RLS in distributed environments. In this section, we present performance results for an LRC using a PostgreSQL relational database back end. For space reasons, we focus on one characteristic of PostgreSQL: the need to perform periodic garbage collection or "vacuum" operations.

In this set of experiments, both the clients and server are workstations in a Linux cluster. Each machine is a dual Pentium-III 547 MHz box with 1

Gigabyte of memory. The OS version is Red Hat Linux 7.2.

In PostgreSQL, when mappings are ostensibly deleted from a table, they are not physically deleted from the disk. A garbage collection or "vacuum" operation must be performed periodically to physically delete them from disks. Vacuum operations are time-consuming and may require exclusive access to the database, preventing other requests from executing.

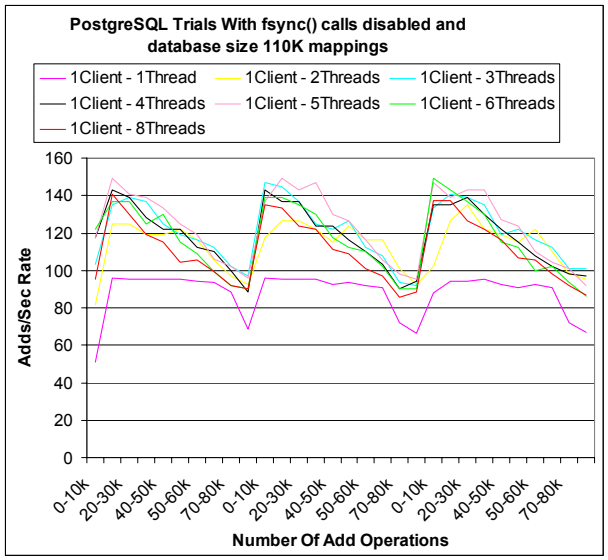

Figure 8: Performance during add and delete tests

Figure 8 shows how the performance of the database is affected when a large number of add and delete operations are performed followed by periodic vacuum operations. The size of the LRC database is 110,000 entries. For each line in the graph, there is one client with one or more threads issuing add operations followed by delete operations. In each trial, 10,000 mappings are added and subsequently deleted. The graph shows a saw-tooth pattern. The add rate decreases steadily as the number of trials (marked by the ranges in the x-axis) increases, until a vacuum operation is performed after 10 trials (or 100,000 operations). After each vacuum operation completes, the add rate returns to its maximum value.

These performance results suggest that in RLS environments with expected high rates of add and delete operations to LRC databases, the garbage collection algorithm for PostgreSQL may significantly limit RLS performance. Under these conditions, MySQL may prove a better choice for the RLS database back end.

5.3 RLI Query Performance

Next, we present the query rates supported by an RLI with a MySQL back end in a 100 megabit per second LAN. The clients for these tests are cluster workstations that are dual Pentium III 547 MHz processors with 1.5 gigabytes of memory running Red Hat Linux version 9. The server is a dual Intel Xeon 2.2 GHz workstation with 1 gigabyte of memory running Debian Linux 3.0.

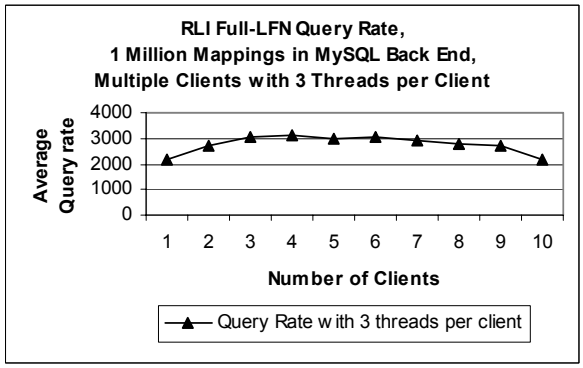

Figure 9: RLI Query Rates with Uncompressed Updates

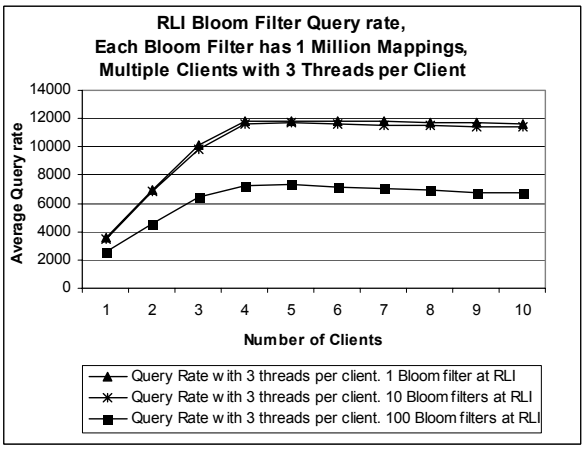

Figure 10: RLI Query Rates with Bloom filter updates

Figure 9 shows query rates of approximately 3000 per second for an RLI that receives full, uncompressed soft state updates. Figure 10 shows much higher query rates for an RLI that receives Bloom filter updates and stores them in memory. This RLI provides similar query rates for one and ten Bloom filters, but the query rate drops for 100 Bloom filters, indicating that the overhead of checking multiple Bloom filter bit maps on a query operation can be significant as the number of LRCs updating the RLI increases.

5.4 Bulk Operations

For user convenience, the RLS implementation includes bulk operations that perform a large number of add, query, or delete operations on mappings or on attributes. Bulk operations are particularly useful for large scientific workflows that perform many RLS query or registration operations. We perform bulk operation tests with 1000 requests per operation. The test configuration is the same as that in the last section.

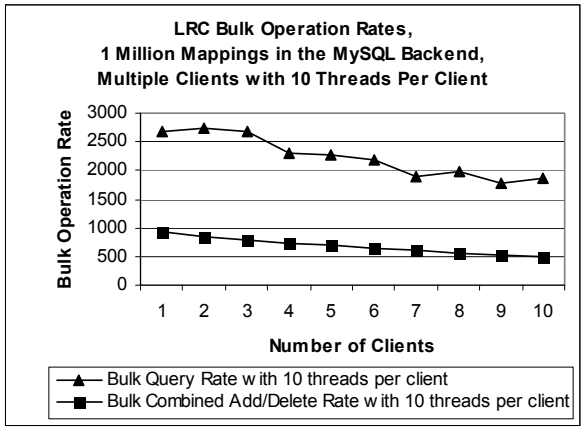

Figure 11: Bulk Operation Rates with 1000 requests per operation.

Figure 11 shows that bulk operations perform better than non-bulk operations by aggregating multiple requests in a single packet to reduce request overhead. The top line shows bulk query rates. The query rate for a single client (10 threads) is 27% higher than the rate achieved by one client performing non-bulk queries in Figure 6. As the total number of threads increases, the performance advantage of bulk queries decreases. For 10 clients (100 threads), bulk queries provide only an 8% improvement in query rates.

The lower line in Figure 11 shows combined add/delete operation rates. To maintain approximately constant database size for this test, each thread issues a bulk operation of 1000 adds followed immediately by a bulk operation of 1000 deletes. The combined bulk add/delete operations perform about 7% better than non-bulk add operations for a single client with 10 threads (Figure 6). For 10 clients (100 threads), bulk add/delete performance is between that of non-bulk add and delete operations (15% worse than non-bulk add rates and 5% better than non-bulk delete rates).

5.4 Uncompressed Soft State Updates

Because the Replica Location Service is hierarchical, one important measure of its scalability is the performance of soft state updates from LRCs to RLIs. Next, we measure the performance of uncompressed soft state updates as LRCs become large and the number of LRCs updating an RLI increases. These tests were conducted in a LAN with 100 megabit per second Ethernet connectivity. Each LRC server sending updates is a node in the Linux cluster described earlier. The RLI server is a dual Intel Xeon 2.2 GHz machine with 1 Gigabyte of memory running Redhat Linux 8. Each server uses a MySQL back end.

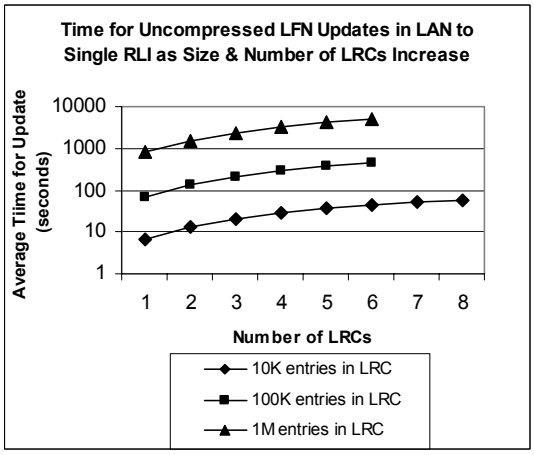

Figure 12: Uncompressed Soft State Update Times

The log-linear scale graph in Figure 12 shows the performance of uncompressed soft state updates as the size of the LRC databases increases from 10,000 to 1 million entries. Update times increase with the size of the LRC database. When multiple LRCs are updating an RLI simultaneously, uncompressed soft state update performance slows dramatically. For example, when 6 LRCs are simultaneously updating the RLI, an average update takes approximately 5102 seconds for an LRC with 1 million entries. These update times are long in a local area network and will show worse scalability in the wide area. The reason for this poor performance is that the rate of updates to an RLI database remains fairly constant as the RLI receives updates from multiple LRCs. Thus, the average time to perform individual soft state updates increases.

These results indicate that performing frequent uncompressed soft state updates does not scale well. Thus, we recommend the use of immediate mode with uncompressed updates or compression to achieve acceptable RLS scalability. Which update mode to deploy may depend on whether applications can occasionally tolerate long full updates and whether they require wildcard searches on RLI contents, which are not possible when using Bloom filter compression.

5.5 Soft State Updates Using Bloom Filter Compression

Next, we measured the performance of soft state updates using Bloom filter compression. These measurements were performed in the wide area, with updates sent from LRCs in Los Angeles to an RLI in Chicago. The mean round trip time was 63.8 milliseconds. The LRC servers for these tests are nodes in the cluster already described. The RLI server is a dual processor Intel Xeon 2.2 GHz machine with 2 gigabytes of memory running Red Hat Linux 7.3. The database used is MySQL. Three hash functions are used to compute the Bloom filter. The Bloom filter size is approximately 10 bits for every LRC mapping.

Table 3: Bloom Filter Update Performance

Database Size (number of mappings)	Avg. Time to Perform Soft State Update (second)	Avg. Time to Generate Bloom Filter (sec)	Bloom Filter Size (bits)
100,000	less than 1	2	1 Million
1 Million	1.67	18.4	10 Million
5 Million	6.8	91.6	50 Million

Table 3 shows Bloom filter update statistics for a single client performing a soft state update for a range of LRC database sizes. The second column shows that Bloom filter wide area updates are significantly faster than uncompressed updates. For example, a Bloom filter update for an LRC with 1 million entries took 1.67 seconds in the WAN compared to 831 seconds for an uncompressed update in the LAN (Figure 12).

The third column in the table shows the time required to 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 unsetting the corresponding bits in the Bloom filter. The fourth column shows the Bloom filter size, which increases with the number of LRC entries.

Next, we demonstrate the scalability of Bloom filter updates in the wide area. For this test, we configured 14 clients as LRCs with databases containing 5 million mappings. Each LRC sends wide area Bloom filter updates continuously (i.e., a new update begins as soon as the previous update completes). In practice, clients are likely to perform updates less frequently than this, so these results show worst-case scalability.

Figure 13 shows that for up to seven clients sending continuous Bloom filter updates, the average client update time remains relatively constant at 6.5 to 7 seconds. As the number of clients increases to 14, the average soft state update time increases to 11.5 seconds, suggesting increasing contention for RLI resources. However, these update times are two to three orders of magnitude better than for uncompressed updates. For example, when 6 LRCs with 1 million mappings perform uncompressed updates to an RLI in Figure 12, the average update time is 5102 seconds in the local area network. In RLS deployments to date, there are typically fewer than 10 LRCs updating an RLI. Bloom filter updates should provide good WAN scalability for such deployments.

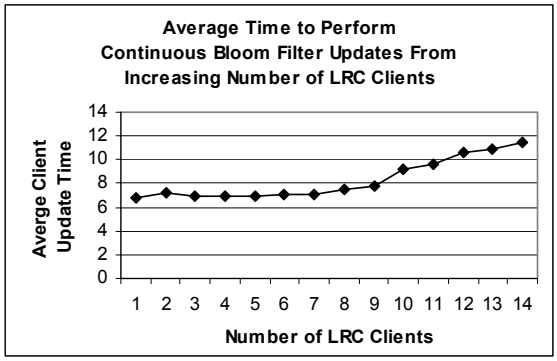

Figure 13: Wide Area Update Scalability

6. RLS Deployments

Several Grid projects are using the Replica Location Service in research or production deployments. These include the LIGO (Laser Interferometer Gravitational Wave Observatory) [7] project, which uses the RLS to register and query mappings between 3 million logical file names and 30 million physical file locations. The Earth System Grid [6] deploys four RLS servers that function as both LRCs and RLIs in a fully-connected configuration and store mappings for 40,000 physical files. The Pegasus system for planning and execution in Grids uses 6 LRCs and 4 RLIs to register the locations of approximately 100,000 logical files [8][9].

7. Ongoing and Future Work

The latest RLS version includes support for a hierarchy of RLI servers that update one another as well as performance and reliability improvements. Through the OGSA Data Replication Services Working Group of the Global Grid Forum [5], we are working to standardize a web service interface for replica location services. A version of RLS based on this interface is planned for Globus Toolkit Version 4.

8. Related Work

Related Grid systems include the Storage Resource Broker [10] and GridFarm [11] projects that register and discover replicas using a metadata service and the European DataGrid Project [12], which has implemented a different Replica Location Service based on the RLS Framework [1].

Also relevant are replication and data location systems for peer-to-peer systems, including Chord, Freenet, Tapestry and OceanStore. Distributed peer-to-peer hash table systems such as Chord [13] and Freenet [14] perform file location and replication by hashing the logical identifiers into keys. Each node is responsible for a subset of the hashed keys and searches for a requested key within its key space, passing the query to a neighbor node "near" in key-space if the key is not found locally. Tapestry [15] nodes form a peer-to-peer overlay network that deterministically associates each data object with a Tapestry location root; this root is used for location purposes. OceanStore [16] employs a two-part data location mechanism that combines a quick, probabilistic search with a slower, guaranteed traversal of a redundant fault-tolerant backing store.

Several distributed file system projects have addressed replication and data location issues. In Ficus [17], collections of file volume replicas are deployed at various storage sites, and a given file may be replicated at any subset of these sites. Bayou [18] is a replicated storage system designed for an environment with variable, intermittent network connectivity. Bayou uses an update-anywhere replication model and a reconciliation scheme.

Mariposa [19] is a distributed database management system that provides asynchronous replica management with relaxed consistency among copies.

9. Summary

We have described the implementation and evaluated the performance of a Replica Location Service included in the Globus Toolkit Version 3.0. Our results demonstrate that individual RLS servers perform well and scale up to millions of entries and one hundred requesting threads. We also demonstrate that soft state updates of the distributed index scale well when using Bloom filter compression.

10. Acknowledgments

This research was supported in part by DOE Coop. Agreements DE-FC02-01ER25449 (SciDAC- DATA) & DE-FC02-01ER25453 (SciDAC-ESG). Work Package 2 of the DataGrid Project co-designed the RLS framework and did extensive performance evaluation of early versions of the RLS. We greatly 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 the RLS.

11. References

[1] A. Chervenak, et. al, "Giggle: A Framework for Constructing Scalable Replica Location Services," Proc. of SC2002 Conf., Baltimore, MD, 2002.

[2] B. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors,” *Comm. of ACM*, 1970. 13(7): 422-426.

[3] L. Fan, et. al, “Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol,” *IEEE/ACM Transactions on Networking*, 2000. 8(3): p. 281-293.

[4] S. Tuecke, et. al, “Open Grid Services Infrastructure (OGSI) Version 1.0”, Global Grid Forum OGSI Working Group, June 27, 2003.

[5] A. Chervenak et. al, “OGSA Replica Location Services”, Global Grid Forum OREP Working Group, Sept. 19, 2003.

[6] The Earth Systems Grid,http://www.earthsystemsgrid.org

[7] E. Deelman, et. al, "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists," 11th Intl. Symp. on High Perf. Distributed Computing, 2002.

[8] E. Deelman, et. al, "Pegasus: Planning for Execution in Grids," GriPhyN Project Technical Report 2002-20, 2002.

[9] E. Deelman, et. al, "Mapping Abstract Complex Workflows onto Grid Environments," *Journal of Grid Computing*, vol. 1, pp. 25-39, 2003.

[10] C. Baru, et. al, "The SDSC Storage Resource Broker," Proceedings CASCON'98 Conference, 1998.

[11] Osamu Tatebe, et. al, "Worldwide Fast File Replication on Grid Datafarm", Proceedings of the 2003 Computing in High Energy and Nuclear Physics (CHEP03), March 2003.

[12] L. Guy, et. al, “Replica Management in Data Grids,” Global Grid Forum 5, 2002.

[13] I. Stoica, et. al, “Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications,” SIGCOMM Conf., 2001.

[14] I. Clarke, et. al, “Protecting Free Expression Online with Freenet,” *IEEE Internet Computing*, Vol. 6, No. 1, 2002.

[15] Ben Y. Zhao, et. al. “Tapestry: A Resilient Global-scale Overlay for Service Deployment,” *IEEE Journal on Selected Areas in Communications*, Vol 22, No. 1, January 2004.

[16] John Kubiatowicz, et. al, “OceanStore: An Architecture for Global-Scale Persistent Storage," Proc. of ASPLOS 2000 Conference, November 2000.

[17] G.J. Popek, et. al, “Replication in Ficus Distributed File Systems,” Workshop on Mgment of Replicated Data, 1990.

[18] D. Terry, et. al, “A Case for Non-Transparent Replication: Examples from Bayou,” *Proc. of IEEE Intl. Conf. on Data Engineering*, pages 12-10, December 1998.

[19] J. Sidell, et. al, “Data Replication in Mariposa”, *12th Intl. Conf. on Data Engineering*. Pages: 485 – 494, 1996.