Giggle:
A Framework for Constructing Scalable Replica Location Services

Ann Chervenak[1] Ewa Deelman[1] Ian Foster[2,3] Leanne Guy[4] Wolfgang Hoschek[4]
Adriana Iamnitchi[2] Carl Kesselman[1] Peter Kunszt[4] Matei Ripeanu[2]
Bob Schwartzkopf[1] Heinz Stockinger[4] Kurt Stockinger[4] Brian Tierney[5]

[1]Information Sciences Institute, University of Southern California, Marina del Rey, CA 90292
[2]Department of Computer Science, University of Chicago, Chicago, IL 60637
[3]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
[4]CERN, European Organization for Nuclear Research, CH-1211 Geneva 23, Switzerland
[5]Lawrence Berkeley National Laboratory

Abstract

In wide area computing systems, it is often desirable to create remote read-only copies (replicas) of files. Replication can be used to reduce access latency, improve data locality, and/or increase robustness, scalability and performance for distributed applications. We define a replica location service (RLS) as a system that maintains and provides access to information about the physical locations of copies. An RLS typically functions as one component of a data grid architecture. This paper makes the following contributions. First, we characterize RLS requirements. Next, we describe a parameterized architectural framework, which we name Giggle (for GIGa-scale Global Location Engine), within which a wide range of RLSs can be defined. We define several concrete instantiations of this framework with different performance characteristics. Finally, we present initial performance results for an RLS prototype, demonstrating that RLS systems can be constructed that meet performance goals.

1 Introduction

In wide area computing systems, it is often desirable to create remote read-only copies (*replicas*) of data elements (*files*). Replication can be used to reduce access latency, improve data locality, and/or increase robustness, scalability and performance for distributed applications. A system that includes replicas requires a mechanism for locating them.

We thus define the *replica location problem*: Given a unique *logical* identifier for desired content, determine the *physical* locations of one or more copies of this content. We further define a *replica location service (RLS)* as a system that maintains and provides access to information about the physical locations of copies.

An RLS typically does not operate in isolation, but functions as one component of a data grid architecture [1-3] (Figure 1). Other components include: (1) the *GridFTP protocol* [4] for secure, efficient wide area data transfer; (2) a *file transfer service* for reliable transfer of files between storage systems; (3) the *RLS*; (4) a *reliable replication service* that provides coordinated, fault-tolerant data movement and RLS updates; (5) a *metadata service* containing information that describes logical files; (6) one or more higher level *data management services* that provide such functionality as version management, master copy management, and workflow management; and

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

(7) *application-oriented data services* that implement application-specific semantics and policies such as preferences for replica selection or access.

In our RLS design we focus on issues of scalability, reliability, and security—concerns that arise in large-scale distributed systems. We target systems that may have tens of millions of data items, tens or hundreds of millions of replicas, hundreds of updates per second, and hundreds or perhaps many thousands of storage systems, and that require high reliability and strong security.

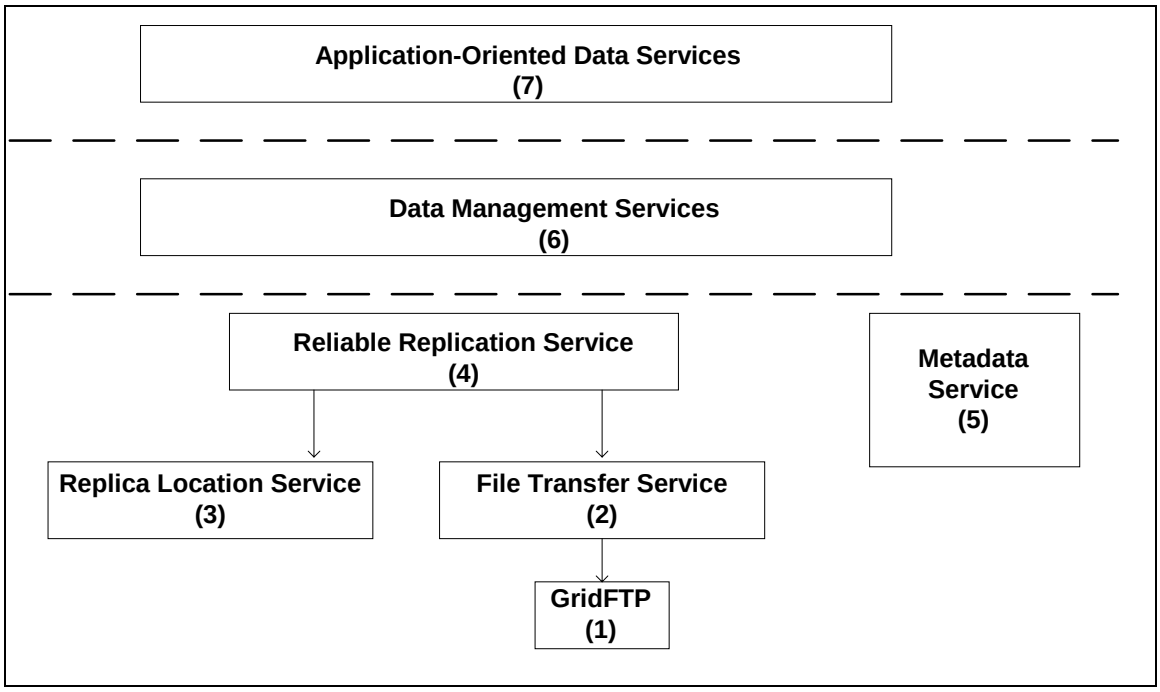

Figure 1: Data Grid Architecture including the Replica Location Service

This paper makes the following contributions to our understanding of Data Grid systems and data replication:

- We introduce the notion of a RLS as a distinct component and characterize RLS requirements
- We describe a parameterized architectural framework, which we name Giggle (for GIGa-scale Global Location Engine), within which a wide range of RLSs can be defined
- We define several concrete instantiations of this framework with different performance characteristics.
- We present initial performance results for an RLS prototype, demonstrating that RLS systems can be constructed that meet performance goals.

2 Requirements for a Replica Location Service

We use the term *logical file name (LFN)* to denote a unique logical identifier for desired data content. The function of an RLS is to identify zero or more physical copies of the content specified by an LFN. Each physical copy is specified by a unique *physical file name* (PFN), such as a GridFTP [4] URL, that specifies its location on a storage system. This concept of a unique logical identifier for a desired data content is applicable only within the context of a virtual organization (VO) [5] that brings together users and resources in the pursuit of common goals.

Discussions with high-energy physics and climate modeling application communities lead us to identify the following RLS requirements:

1. *Read-only data and versioning:* Files do not change or change only infrequently and can be uniquely identified as distinct versions. While these assumptions do not apply universally, they characterize a large class of data-intensive applications. For example, in many scientific collaborations, data are prepared, annotated, and then published to the community. After this act of publication, files are immutable.
2. *Size:* The system must scale to at least several hundred replica sites, 50 million logical files and 500 million total physical files or replicas.
3. *Performance:* The system must handle up to 1000 queries and 200 updates per second. Average response time should be below 10 milliseconds, and maximum query response time should not exceed 5 seconds.
4. *Security:* The RLS is most concerned with protecting the privacy and integrity of knowledge about the existence and location of data, while individual storage systems protect the privacy and integrity of data content.
5. *Consistency:* An RLS need not provide a completely consistent view of all available replicas, in the following sense: if an RLS query returns to an authorized client only a subset of all extant replicas, or returns a list of replicas that includes "false positives" (i.e., putative replicas that do not in fact exist), the requesting client may execute less efficiently, but will not execute incorrectly.
6. *Reliability:* The RLS should not introduce a single point of failure such that if any site fails or becomes inaccessible, the entire service is rendered inoperable. In addition, local and global state should be decoupled so that failure or inaccessibility of a remote RLS component does not affect local access to local replicas.

3 The Giggle Replica Location Service Framework

The framework that we have designed to meet the requirements listed above is based on the recognition that different applications may need to operate at different scales, have different resources, and have different tolerances to inconsistent replica location information. Thus, its design allows users to make tradeoffs among consistency, space overhead, reliability, update costs, and query costs by varying six simple system parameters (described in detail in Table 1). The Giggle framework is structured in terms of five basic mechanisms:

- *Independent local state maintained in Local Replica Catalogs (LRCs)*
- *Unreliable collective state maintained in Replica Location Indices (RLIs)*
- *Soft state maintenance of RLI state*
- *Compression of state updates*
- *Membership and partitioning information maintenance*

3.1 Local Replica Catalogs

A *local replica catalog* (LRC) maintains information about replicas at a single replica site. An LRC must meet the following requirements:

- *Contents*. It must maintain a mapping between arbitrary logical file names (LFNs) and the physical file names (PFNs) associated with those LFNs on its storage system(s).
- *Queries*. It must respond to the following queries:
    - *Given an LFN, find the set of PFNs associated with that LFN.*
    - *Given a PFN, find the set of LFNs associated with that PFN.*
- *Local integrity*. It is the responsibility of the LRC to coordinate the contents of the name map with the contents of the storage system in an implementation-specific manner.
- *Security*. Information within the LRC may be subject to access control, and therefore must support authentication and authorization mechanisms when processing remote requests.
- *State propagation*. The LRC must periodically send information about its state using a state propagation algorithm, *S*, to RLI(s), as discussed in the next two subsections.

3.2 The Replica Index

While the various LRCs collectively provide a complete and locally consistent record of all extant replicas, they do not directly support user queries about multiple replica sites. An additional *index structure* is required to support these queries.

The Giggle framework structures this index as a set of one or more Replica Location Indices (RLIs), each of which contains a set of (LFN, pointer to an LRC) entries. A variety of index structures can be defined with different performance characteristics, simply by varying the number of RLIs and amount of redundancy and partitioning among the RLIs (see Figure 2 and Figure 3).

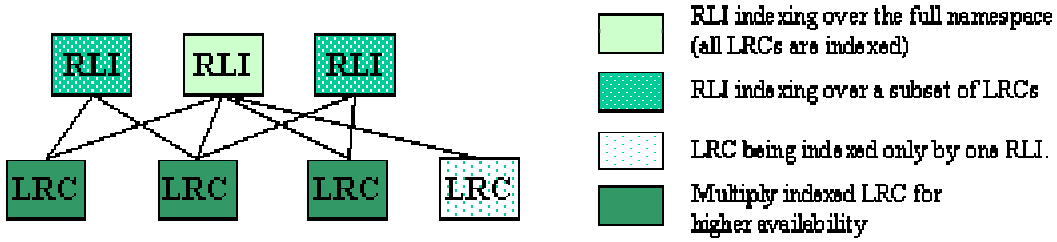

Figure 2: A 2-level RLS layout.

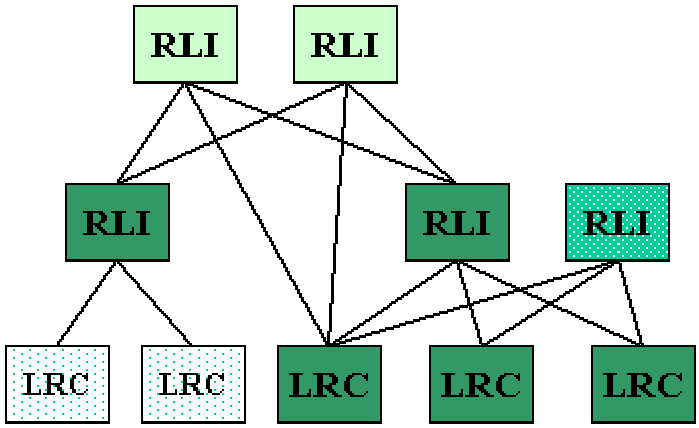

Figure 3: A hierarchical RLS topology. Note that the two top-level nodes may be redundant and hold exactly the same mappings. Alternatively, as explained in the text, we may partition state updates from LRCs to LRIs according to the namespace. In that case the two top-level nodes index different portions of the LFN namespace.

Table 1: The six parameters used to characterize Giggle RLS structures, and some examples of possible values and their implications. See text for more details.

G	**The number of RLIs**	
	G=1	A centralized, non-redundant or partitioned index
	G>1	An index that includes partitioning and/or redundancy
	G ≥ N	A highly decentralized index with much partitioning and/or redundancy
P_L	**The function used to partition the LFN name space**	
	$P_L = \phi$	No partitioning by LFN. The RLI(s) must have sufficient storage to record information about all LFNs, a potentially large number
	P_L=hash	"Random" partitioning. Good load balance, perhaps poor locality
	P_L=coll	Partitioning on collection name. Perhaps poor load balance, good locality
P_R	**The function used to partition the replica site name space**	
	$P_R = \phi$	No partitioning by site name. Indices have entries for every replica of every LFN they are responsible for. Potentially high storage requirements
	P_R=IP	Partition by domain name or similar. Potential geographic locality
R	**The degree of redundancy in the index space**	
	R=1	No redundancy: each replica is indexed by only one RLI
	R=G>1	Full index of all replicas at each RLI. Implies no partitioning, much redundancy/space overhead
	1<R<G	Each replica indexed at multiple RLIs. Less redundancy/space overhead

C	**The function used to compress LRC information prior to sending**	
	C= ϕ	No compression: RLIs receives full LFN/site information
	C=bloom	RLIs receive summaries with accuracy determined by Bloom parameters
	C=coll	RLIs receive summaries based on collection distribution
S	**The function used to determine what LRC information to send when**	
	S=full	Periodically send entire state (perhaps compressed) to relevant RLIs
	S=partial	In addition, send periodic summaries of updates, at a higher frequency.

We can characterize a wide range of global index structures in terms of six parameters (G, P_L, P_R, R, S, C). As summarized in Table 1, four parameters (G, P_L, P_R, R) describe the distribution of replica information to RLIs and two define how information is communicated from LRCs (S, C). The parameter G specifies the total number of RLIs in the replica location service. P_L determines the type of logical file name space partitioning of information sent to the RLIs. The parameter P_R indicates the type of LRC name space partitioning by which a particular RLI receives state updates from only a subset portion of all LRCs. R indicates the number of redundant copies of each RLI maintained in the replica location service. The soft state algorithm S indicates the type and frequency of updates sent from LRCs to RLIs. Finally, the parameter C indicates whether a compression scheme is used to reduce the size of soft state updates.

Based on the redundancy, partitioning, and soft state mechanisms possible in the Giggle framework, we summarize requirements that must be met by a global replica index node (RLI).

- *Secure remote access:* An RLI must support authentication, integrity and confidentiality and implement local access control over its contents.
- *State propagation.* An RLI must accept periodic inputs from LRCs describing their state. If the RLI already contains an LFN entry associated with the LRC, then the existing information is updated or replaced. Otherwise, the index node creates a new entry.
- *Queries.* It must respond to queries asking for replicas associated with a specified LFN by returning information about that LFN or an indication that the LFN is not present in the index.
- *Soft state.* An RLI must implement time outs of information stored in the index. If an RLI entry associated with an LRC has not received updated state information from the LRC in the specified time out interval, the RLI must discard the entries associated with that LRC.
- *Failure recovery.* An RLI must contain no persistent replica state information. That is, it must be possible to recreate its contents following RLI failure using only soft state updates from LRCs.

3.3 Soft State Mechanisms and Relaxed Consistency

We have argued that strong consistency is not required in the RLS, allowing us to use a *soft state* protocol [6-9] to send LRC state information to relevant RLIs, which then incorporate this information into their indices.

Soft state is information that times out and must be periodically refreshed. There are two advantages to soft state mechanisms. First, stale information is removed implicitly, via time outs, rather than via explicit delete operations. Hence, removal of data associated with failed or

inaccessible replica sites can occur automatically. Second, RLIs need not maintain persistent state information, since state can be reconstructed after RLI failures using the periodic soft state updates from LRCs. Various soft state update strategies with different performance characteristics can be defined.

3.4 Compression

Next, we consider compression of the soft state information communicated from an LRC to RLI(s). "Uncompressed" updates communicate all LFNs located at the LRC and allow the RLI(s) to maintain an index that is accurate, modulo time delays between when changes occur at the LRC and when updates are processed at the RLI(s). To reduce network traffic and the cost of maintaining RLIs, we can also compress LFN information in various ways, for example:

- Using hash digest techniques such as Bloom filters [10, 11].
- Using structural or semantic information in LFNs, such as the names of logical collection that contain a group of logical files.

3.5 Membership and Partitioning Information Maintenance

The set of LRCs and RLIs that compose the RLS changes over time as components fail and new components join the distributed system. In this section, we describe the mechanisms that LRCs and RLIs use to perform *service discovery* to locate other services in the RLS. We also discuss how policies regarding the partitioning of the index among RLIs are specified and maintained. These mechanisms are compatible with the emerging Open Grid Services Architecture [12].

Each LRC and RLI server maintains self-descriptive metadata. At deployment time, each server is configured by the virtual organization with information about one or more components in the system. At bootstrap, each component sends its service metadata to these known locations. The RLS components can then use some service discovery mechanism to obtain information about other components (e.g., [13, 14]). In effect, each RLI acts as a registry, collecting information on LRCs and RLIs known to it and answering queries about them. An RLS client can discover the existence and service description of an RLI or LRC either statically, as part of the VO configuration, or by using a service registry technology.

For a partitioned RLS, each RLS server must also discover the policies that define system behavior. These policies may be initialized at deployment time, but repartitioning may be required as servers fail or enter the RLS. One promising approach is to use consistent hashing [15, 16] for both namespace and replica site partitioning. Consistent hashing is a load-balanced, low-overhead mapping technique from a domain A to a codomain B that requires a minimal number of changes when the codomain changes. In our context, the significance is that only a small number of LRCs that are directly affected by the departure of an RLI will need to reconnect to different RLIs.

4 Implementation Approaches

We now illustrate the application of the Giggle framework by describing five RLS implementation approaches.

RLS0. *Single RLI for all LRCs*. Each LRC sends its full, uncompressed state to a single RLI. The obvious disadvantage of this scheme is its single point of failure. The six Giggle framework parameters (Table 1) in this case are: G=R=1 (and hence P_L=P_R=ϕ), S=all, C=ϕ.

RLS1. *LFN Partitioning, Redundancy, Bloom Filters*. Figure 4 shows an RLS implementation that includes redundancy, with two copies of each index. Indexed

information is partitioned based on the logical file namespace. Soft state updates are compressed using Bloom filters to summarize LRC state. The Giggle parameters are: G>1, R=2, PL=partition-on-LFN, PR=ϕ, S=partial, C=Bloom filtering.

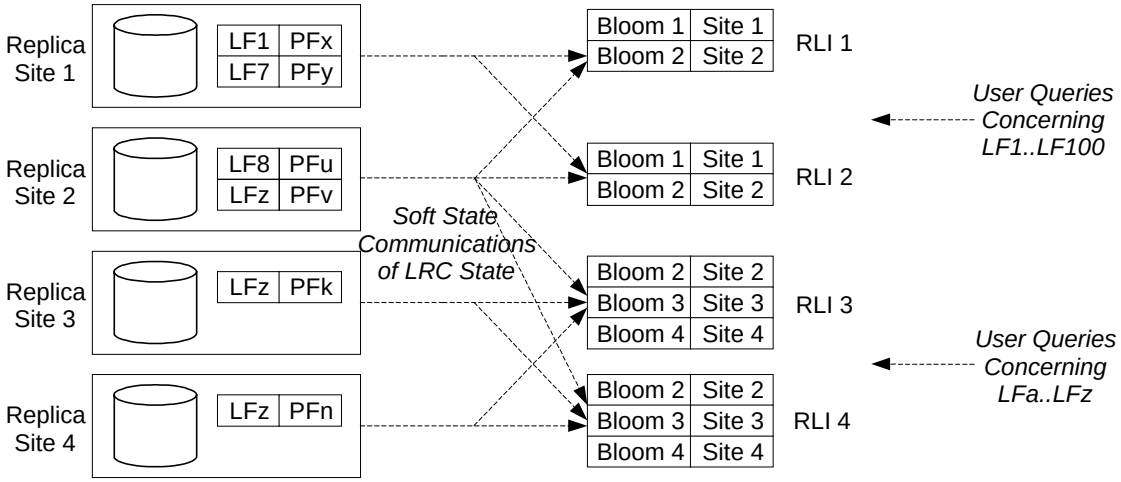

Figure 4: A possible configuration of an RLS2 implementation.

RLS3. *Compression, Partitioning based on Collections.* Figure 5 shows an implementation that includes both compression and partitioning of the logical file namespace based on domain-specific logical collections. The framework parameters are as follows: *G>1*, *R*=2, P_L=coll, P_R=ϕ, *S*=full, *C*=collection-name-based.

RLS4. *Replica Site Partitioning, Redundancy, Bloom Filters*. Storage partitioning was shown in Figure 1. That configuration represents the following Giggle parameters: *G*=3, *R*=1, P_L=ϕ, P_R=IP, *S*=full, *C*=unspecified.

RLS5. *A Hierarchical Index*. Multiple levels of RLIs may be included in an RLS (as shown in Figure 3). Thus, the Giggle framework supports a wide variety of RLS configurations that can be adapted dynamically for increased scalability, performance and reliability.

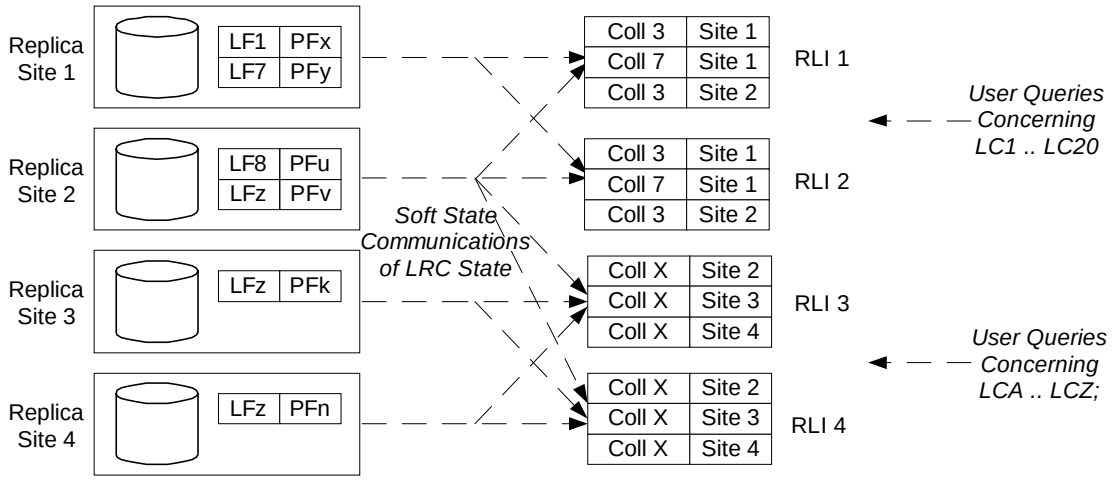

Figure 5: Structure of RLS3, which uses collection names for partitioning function.

5 RLS Prototype Implementation

Figure 6 shows the components of our RLS prototype, which we have implemented in C and tested on Solaris 8, Linux RH 6.1, 6.2 and 7.2. This prototype relies on Grid Security Infrastructure and the globus_io socket layer from the Globus toolkit [17] to provide a server front end to a relational database. The server is multithreaded and can be configured to act as an LRC and/or an RLI. Clients access the server via a simple RPC protocol that will soon be replaced by a SOAP interface. For convenient replacement of the relational database backend, our implementation includes an ODBC (Open Database Connectivity) layer, an API for database access. In addition to the libiodbc library, our implementation includes a myodbc layer that maps from ODBC to the mySQL database, which is used as the relational database backend for our prototype. The database contains several tables that implement {lfn, pfn} mappings.

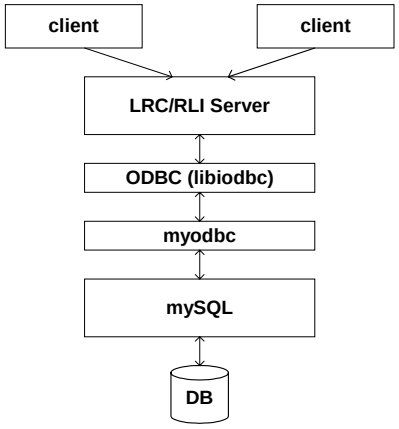

Figure 6: The layered design of the LRC/RLI server.

The server's configuration includes information such as LRC soft state update frequency and timeout information for RLI entries. We have implemented complete and partial soft state updates from LRCs to RLIs and partitioning of the logical file namespace based on path name prefixes.

At present, the prototype uses a static configuration for service discovery and maintenance of partitioning information. The prototype does not currently implement compression.

6 Performance of the RLS Prototype

We present preliminary performance results for our prototype. We focus on two issues: the performance of basic operations on each server and the overhead of soft state updates between LRCs and RLIs.

Figure 7 shows the performance of create, add, delete and query operations on an LRC server. A create operation defines an {lfn, pfn} mapping for a logical file name that is not currently registered in the RLS, while an add operation registers an additional mapping for an existing LFN. The delete operation deletes a single {lfn, pfn} mapping. The graph shows average operation times for different LRC database sizes computed over 5000 sequential operations. These measurements were run on a machine with dual 2.2 GHz processors and 1 GB of memory running RedHat Linux 7.2. We achieved rates of 1667 queries per second and approximately 67 updates (creates, adds or deletes) per second when operations are issued from a single client thread. For the database sizes studied (up to 1 million entries), these operation times were relatively constant but did not include network latencies.

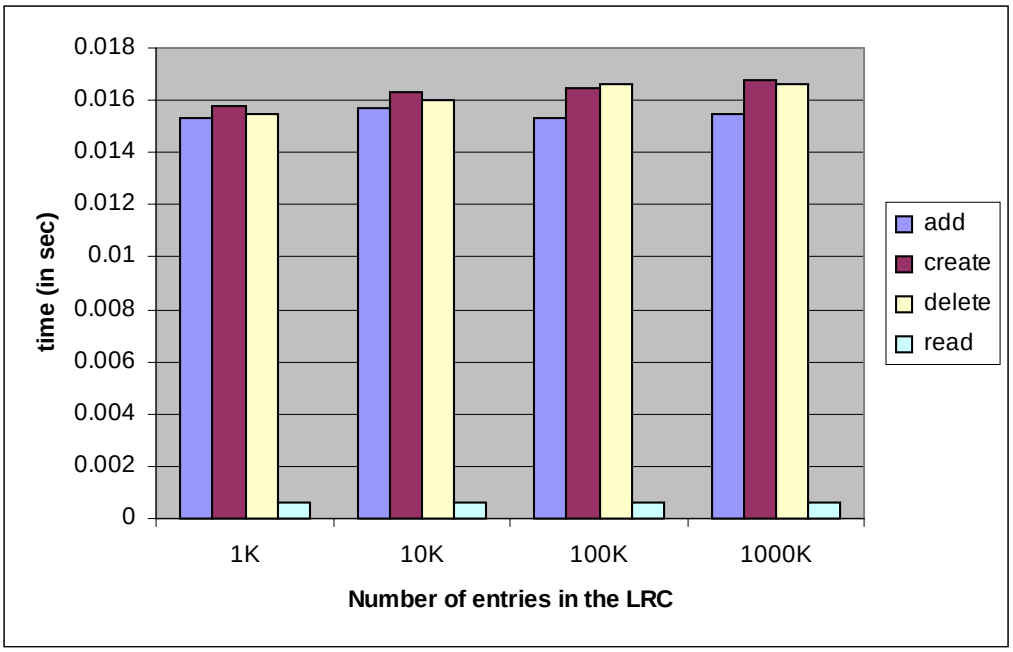

Figure 7: Time to create, delete, add and query a LFN entry in LRC and to query a RLI.

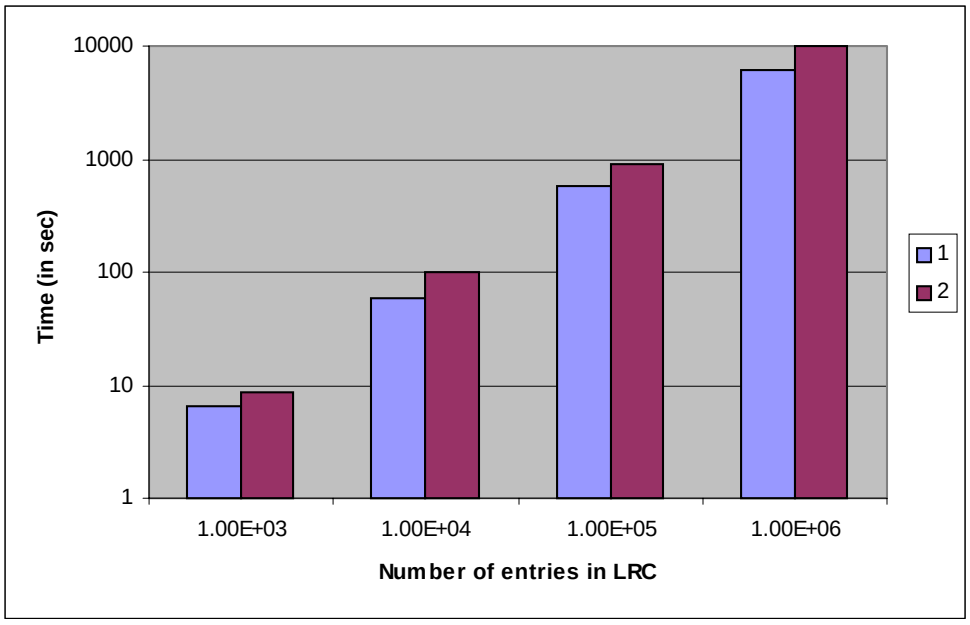

Figure 8: Time to for an LRC to send a complete soft state update to an RLI when there are one or two LRCS sending simultaneous updates.

Figure 8 shows the time to complete a soft state update from the perspective of the RLI when one or two LRCs are updating the RLI on a local area network. These soft state update timings are linear with the size of the LRC. For an LRC with 1 million entries, the soft state update time is 6217 seconds, or 1.73 hours. When two LRCs simultaneously update an RLI, the time to complete one LRC update increases by approximately 50% due to concurrent updates on the RLI. These update tests measure the time for an LRC running on the dual-processor machine described above to update an RLI on a Sun blade 1000 with dual 750 MHz processors and 1 gigabyte of RAM running Solaris 8. The second LRC in these tests is a slower Sun Ultra 5_10 running Solaris 2.6 with 128 K of RAM.

Figure 9 shows the timings for complete soft state updates in the wide area. These tests were run on the European DataGrid testbed. The tests show up to five LRCs (three in Geneva and two in Pisa) sending soft state updates to a single RLI located in Glasgow. The graph shows that wide area complete soft state update times increase both with the size of the LRCs and the number of LRCs sending updates.

Both Figure 8 and Figure 9 show that using complete soft state updates is potentially quite slow and is not likely to scale well in production settings. These results clearly indicate the need for techniques such as compression and/or incremental updates, in which only part of the LRC state information is sent to an RLI.

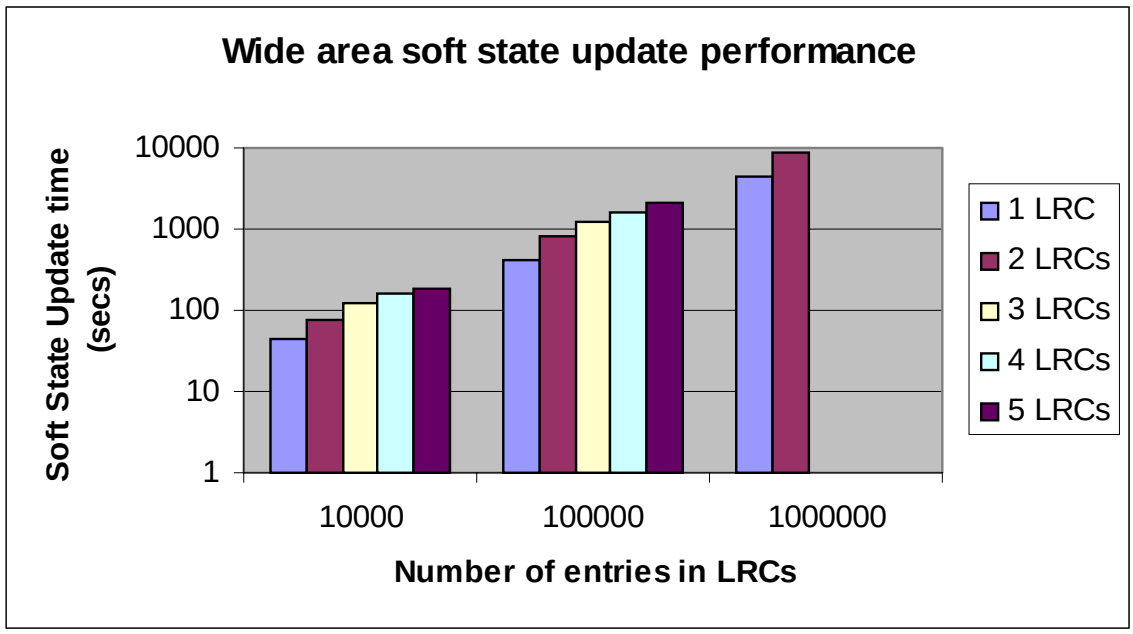

Figure 9: Wide Area Complete Soft State Update Performance.

In the following set of experiments, we evaluate the performance of one LCR server updating a single RLI server using full state updates only or a combination of full state updates and incremental updates. In the incremental case, updates are performed after 200 update operations are executed on the server or after a time interval of 30 seconds has elapsed, whichever occurs first.

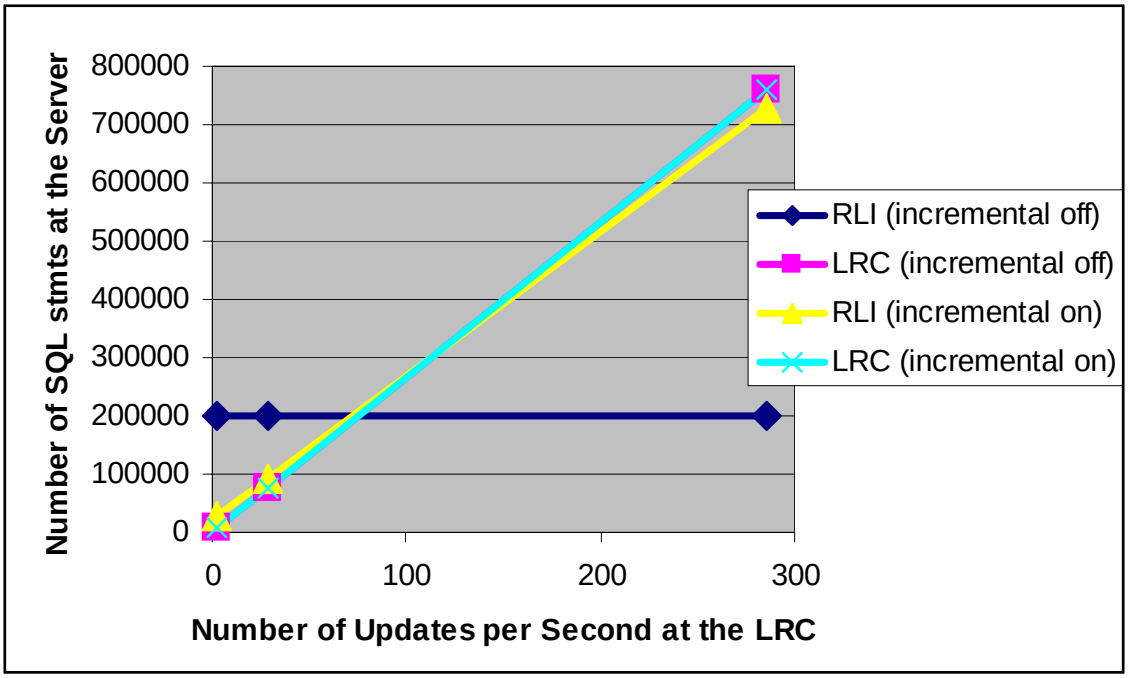

Figure 10: Number of SQL Operations Generated at the RLI and LRC Servers for Complete and Incremental Updates.

In Figure 10, we show the number of SQL operations generated on the LRC and RLI when the size of the LRC is 10000 entries. With both types of soft state updates, the number of SQL operations on the LRC increases linearly with the update rate of LRC entries. In the case where only complete soft state updates are sent to the RLI (labeled "incremental off" in the graph), the

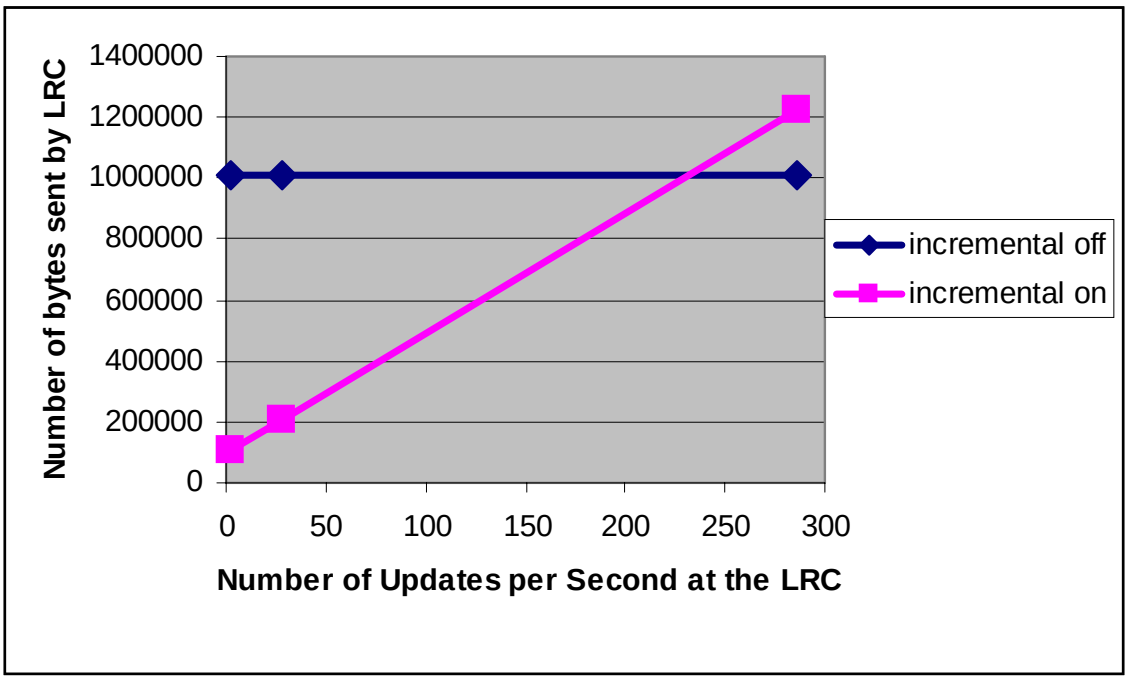

Figure 11: Number of Bytes Sent by the LRC to the RLI for Complete Soft State Updates and for Combined Incremental and Soft State Updates.

number of SQL operations to update the RLI is constant with respect to the update rate of entries in the LRC, since the size of complete updates corresponds to the total size of the LRC. When incremental updates are used in combination with occasional complete updates, we see fewer SQL statements at the RLI for lower update rates on the LRC, since only recently updated information is sent during incremental updates. However, as update rates on the LRC increase, the benefit of performing incremental updates decreases, and for sufficiently high update rates, incremental updates generate more SQL overhead on the RLI than complete updates.

In Figure 11, we show the total number of bytes sent from the LRC to the RLI for the two updates schemes. Fewer bytes are sent in the combined incremental and full state update scheme when the number of updates the LRC receives from clients is small. However, when the rate of updates increases, the amount of data sent between the servers increases linearly and eventually surpasses the amount of data sent in the complete update scenario.

The above results show that when the RLS is deployed, the servers need to be configured (statically or dynamically) to use the update scheme that is the most appropriate for the expected rate of updates to the LRC.

Finally, we present some results for partitioning the RLI index space based on pattern matching of logical file names. Recall that partitioning should reduce the number of bytes sent to each RLI and the number of SQL operations that each RLI needs to perform, at the cost of increased CPU overhead at the LRC to determine which RLI should receive a particular update.

Figure 12 shows that the CPU overhead at the LRC is directly proportional to the number of patterns that the LRC must check to determine the receiving RLI. In practice the number of patterns will depend on the logical filename structure and the granularity of the RLI partitioning.

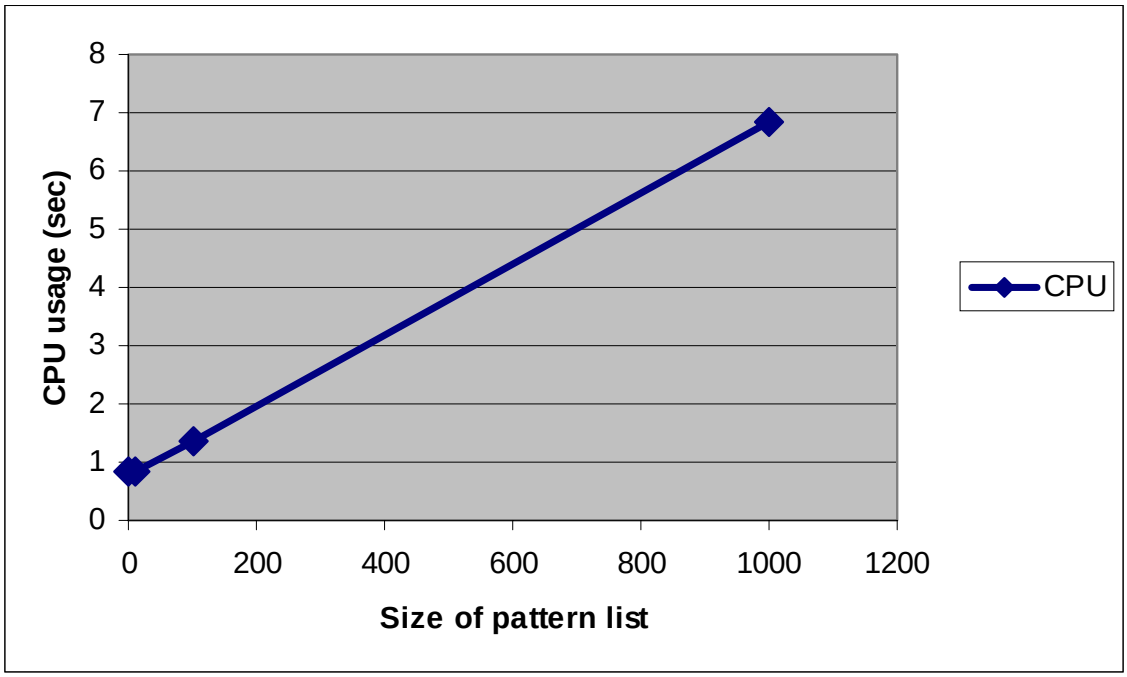

Figure 12: CPU Overhead at the LRC to Perform Pattern Matching for 100000 LFNs, vs. the Number of Patterns (for Partitioned RLI Space)

We are evaluating additional partitioning methods in addition to our current implementation that uses regular expression matching. We also envisage that communities might provide their own partitioning schemes based on their application-specific logical file namespaces.

7 Related Work

Much research has been done on data replication in distributed file systems and databases [18-29]. A primary emphasis of that work was maintaining consistency and resolving conflicts among replicated data. Our work differs from these systems in restricting its focus to maintaining a distributed registry of replicas. Our RLS framework is flexible and can easily be tuned to the needs of a specific class of applications. The RLS can be used as a stand-alone service or as part of a larger data grid architecture that may provide higher level services, including a consistent replication service.

Cooperative Internet proxy-caches [30] offer similar functionalities to our RLS. Hierarchical caching in proxy servers has been extensively analyzed [31, 32]. Two distinct solutions that do not use hierarchies are Summary Cache [11] and the Cache Array Routing Protocol [33, 34].

Service and resource discovery have been a topic of extensive research. The most relevant to our current work are resource discovery systems in which resources are uniquely identified by an attribute (usually a globally unique name): CAN [35], Chord [36], Tapestry [37], Gnutella [38], Freenet [39], and Napster. An exception is Ninja's Service Discovery Service [40, 41], in which services, identified by sets of attributes, are given a "name" built as a Bloom filter on a subset of attributes.

Soft state techniques are used in various Internet services, for example RSVP [7, 9]. The Globus Toolkit's Monitoring and Discovery Service (MDS-2) [42] and the proposed WSDA Hyper Registry [43] use soft state concepts to propagate information about the existence and state of Grid resources.

We previously developed a *replica catalog* service [4] designed to provide a consistent view of replica location. Our initial implementation used a centralized catalog and is integrated with replica management tools that reliably create and delete replicas. This service has seen extensive use (e.g., [44]), but has scalability limitations. The Storage Resource Broker [45] offers a variety of replica creation and selection options that are managed through a logically centralized metadata catalog.

8 Summary

We have proposed Giggle, a flexible framework for constructing scalable replica location services in wide area environments. This framework allows us to tune the behavior of the RLS system based on the scale, performance, reliability and cost requirements of particular classes of applications. Initial performance results for a prototype implementation of a replica location service instantiation show excellent scalability within the local replica catalog component. These results also demonstrate the advantages of techniques for incremental soft state updates, compression and partitioning to reduce update costs on replica location index nodes. Our initial performance results are promising, and we will continue to evaluate RLS performance and to study various algorithms for state updates, compression and partitioning.

9 Acknowledgements

We are grateful to colleagues for discussions on the topics covered in this paper, in particular Andy Hanushevsky, Koen Holtman, Laura Pearlman, Mei-Hui Su, Von Welch, and Mike Wilde. This research was supported in part by the National Science Foundation under contract ITR-

0086044 (GriPhyN), by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38 (Data Grid Toolkit), and by the European Union by the DataGrid project. This material is based upon work supported by the National Science Foundation under Cooperative Agreement No. AST-0122449.

References

1. *The DataGrid Architecture*. 2001, EU DataGrid Project.
2. Chervenak, A., et al., *The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets.* J. Network and Computer Applications, 2001(23): p. 187-200.
3. Foster, I. and C. Kesselman, *A Data Grid Reference Architecture*. 2001.
4. Allcock, W., et al., *Data Management and Transfer in High-Performance Computational Grid Environments.* Parallel Computing, 2001.
5. Foster, I., C. Kesselman, and S. Tuecke, *The Anatomy of the Grid: Enabling Scalable Virtual Organizations.* International Journal of High Performance Computing Applications, 2001. **15**(3): p. 200-222.
6. Chandy, K.M., A. Rifkin, and E. Schooler, *Using Announce-Listen with Global Events to Develop Distributed Control Systems.* Concurrency: Practice and Experience, 1998. **10**(11-13): p. 1021-1027.
7. Clark, D.D. *The Design Philosophy of the DARPA Internet Protocols*. in *SIGCOMM Symposium on Communications Architectures and Protocols*. 1988: ACM Press.
8. Raman, S. and S. McCanne, *A Model, Analysis, and Protocol Framework for Soft State-based Communication.* Computer Communication Review, 1999. **29**(4).
9. Zhang, L., et al. *RSVP: A new Resource ReSerVation Protocol*. in *IEEE Network.* 1993.
10. Bloom, B., *Space/Time Trade-offs in Hash Coding with Allowable Errors.* Communications of ACM, 1970. **13**(7): p. 422-426.
11. Fan, L., et al., *Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol.* IEEE/ACM Transactions on Networking, 2000. **8**(3): p. 281-293.
12. Foster, I., et al., *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. 2002, Globus Project.
13. Hoschek, W., *A Unified Peer-to-Peer Database Framework and its Application for Scalable Service Discovery*, in *2002*. 2002, CERN.
14. Iamnitchi, A. and I. Foster. *On Fully Decentralized Resource Discovery in Grid Environments*. in *International Workshop on Grid Computing*. 2001.
15. Karger, D.R., et al. *Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*. in *Symposium on Theory of Computing*. 1997: ACM.
16. Karger, D.R., et al., *Web Caching with Consistent Hashing.* Computer Networks, 1999. **31**(11-16): p. 1203-1213.
17. Foster, I. and C. Kesselman, *Globus: A Toolkit-Based Grid Architecture*, in *The Grid: Blueprint for a New Computing Infrastructure*, C. Kesselman, Editor. 1999, Morgan Kaufmann. p. 259-278.
18. Braam, P.J., *The Coda Distributed File System.* Linux Journal, 1998. **50**.
19. Breitbart, Y. and H. Korth. *Replication and Consistency: Being Lazy Helps Sometimes*. in *16th ACM Sigact/Sigmod Symposium on the Principles of Database Systems*. 1997. Tucson, AZ.
20. Düllmann, D., et al. *Models for Replica Synchronisation and Consistency in a Data Grid*. in *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)*. 2001. San Francisco, California.

21. Gray, J., et al. *The Dangers of Replication and a Solution.* in *ACM SIGMOD Conference*. 1996.
22. Petersen, K., et al. *Flexible Update Propagation for Weakly Consistent Replication*. in *16th ACM Symposium on Operating Systems Principles (SOSP-16)*. 1997. Saint Malo, France.
23. Popek, G.J., et al. *Replication in Ficus distributed file systems.* in *Workshop on Management of Replicated Data,*. 1990: IEEE.
24. Sidell, J., et al. *Data Replication in Mariposa*. in *12th International Conference on Data Engineering*. 1996. New Orleans, LA.
25. Stockinger, H. *Distributed Database Management Systems and the Data Grid.* in *18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*. 2001. San Diego, CA.
26. Stonebraker, M., et al., *Mariposa: A Wide-Area Distributed Database System.* VLDB Journal, 1996. **5**(1): p. 48-63.
27. Terry, D.B., et al. *The Case for Non-transparent Replication: Examples from Bayou*. in *IEEE Data Engineering*. 1998.
28. Thomas W. Page, J., et al. *Management of replicated volume location data in the Ficus replicated file system.* in *USENIX Conference Proceedings.* 1991.
29. Wiesman, M., et al. *Database Replication Techniques: A Three Paramater Classification.* in *19th IEEE Symposium on Reliable Distributed Systems*. 2000. Nuernberg, Germany,.
30. Wolman, A., et al. *On the scale and performance of cooperative Web proxy caching*. in *Proceedings of 17th ACM Symposium on Operating Systems Principles (SOPS'99)*. 1999. Kiawah Island Resort, SC, USA.
31. Wang, J. *A Survey of Web Caching Schemes for the Internet*. in *Proceedings of ACM SIGCOMM '99 Conference*. 1999.
32. Yu, P.S. and E.A. MacNair. *Performance study of a collaborative method for hierarchical caching in proxy servers*. in *Proceedings of 7th International World Wide Web Conference (WWW7)*. 1998.
33. Ross, K.W., *Hash routing for collections of shared Web caches.* IEEE Network, 1997: p. 37-44.
34. Valloppillil, V. and K.W. Ross. *Cache array routing protocol v1.0*. in *Internet Draft*. 1988.
35. Ratnasamy, S., et al. *A Scalable Content-Addressable Network*. in *SIGCOMM Conference*. 2001: ACM.
36. Stoica, I., et al. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. in *SIGCOMM Conference*. 2001: ACM.
37. Zhao, B.Y., J.D. Kubiatowicz, and A.D. Joseph, *Tapestry: An infrastructure for fault-tolerant wide-area location and routing.* 2001, UC Berkeley.
38. Ripeanu, M., I. Foster, and A. Iamnitchi, *Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design*. 2001, University of Chicago.
39. Clarke, I., et al. *Freenet: A Distributed Anonymous Information Storage and Retrieval System*. in *ICSI Workshop on Design Issues in Anonymity and Unobservability*. 1999.
40. Czerwinski, S.E., et al. *An Architecture for a Secure Service Discovery Service*. in *Mobicom '99*. 1999: ACM Press.
41. Hodes, T.D., et al., *An Architecture for Secure Wide-Area Service Discovery.* Wireless Networks, 2001.
42. Czajkowski, K., et al. *Grid Information Services for Distributed Resource Sharing*. in *10th IEEE International Symposium on High Performance Distributed Computing*. 2001: IEEE Press.

43. Hoschek, W. *A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery*. in *International IEEE Symposium on Parallel and Distributed Computing*. 2002.
44. Stockinger, H., et al. *File and Object Replication in Data Grids*. in *10th IEEE Intl. Symp. on High Performance Distributed Computing*. 2001: IEEE Press.
45. Baru, C., et al. *The SDSC Storage Resource Broker*. in *Proc. CASCON'98 Conference*. 1998.