Making the Best of a Bad Situation: Prioritized Storage Management in GEMS

J. M. Wozniak [a] P. Brenner [a] D. Thain [a] A. Striegel [a]
J. A. Izaguirre [a]

[a] *Dept. of Computer Science & Engineering*
University of Notre Dame
Notre Dame, IN 46556 USA
{ jwozniak, pbrenne1, dthain, striegel, izaguirr } @ nd.edu

Abstract

As distributed storage systems grow, the response time between the occurrence of a fault, detection, and repair becomes significant. Systems built on shared servers have additional complexity because of the high rate of service outages and revocation. Managing high replica counts in this environment becomes very costly in terms of the storage required and bandwidth consumption for file copies. The storage challenge for this situation can thus be phrased as an attempt to function inexpensively with respect to cost constraints such as: disk utilization, network bandwidth consumption, and server CPU time. The GEMS (Grid Enabled Molecular Simulation) [1] storage system provides a replicated and shared workspace for large scale molecular dynamics simulations, and exemplifies the above issues. The framework offers a solution to this problem by prioritizing observed faults and repairing them in an intelligent manner. In this paper, we provide observations from the operation of GEMS and compare its error handling to like storage systems.

Key words: replicated, shared, simulation, storage

1 Introduction

An important observation about large systems is that their response to user and internal system operations is delayed, scaling with the size of the system. The exact nature of the scaling, of course, depends on the algorithms used. In an active replication system, system operations to maintain integrity must

[1] This work was supported in part by the National Science Foundation through the grant NSF DBI-0450067.

become more frequent as the system grows, and often more expensive individually. This is true even if we assume that the underlying components do not become less reliable, which is a dubious assumption in grid computing. A typical solution to increase system robustness is to decrease the reliance on a central server, either by deploying redundant central servers or by shifting the system architecture to an independent, peer-to-peer model. Yet, reducing the reliance on a centralized server tends to increase the number of computers involved in a given transaction, thus again increasing latency. This architectural technique also increases the algorithmic complexity of system operations.

When systems are decentralized, fault-tolerance must be built in. An additional complexity is often added to grid enabled storage systems: the distributed ownership of the resource fabric. A shared system allows providers to pull the plug on their volunteered system at any moment. Thus, such a system incorporates as little confidence in a storage device as possible, and fault tolerance is a foundational design feature.

What do we mean by low confidence? A volunteered resource, such as a commodity desktop workstation, may be rebooted several times a day. However, this does not directly indicate permanent data loss, as the vast majority of machines come back online after a reboot. It does mean that its data is temporarily unavailable, and that an alternative replica must be available. Even users of mature storage systems, as discussed in the following sections, must sometimes wait for data. However, a properly replicated system will nearly eliminate such latencies.

Many modern data grid storage systems are in use for scientific projects [Finkelstein et al., 2004], and those that utilize data replication may be categorized as static or dynamic replica systems [Venugopal et al., 2006]. While static systems place replicas once, possibly with direct user control, dynamic systems constantly probe for faults and repair them with the allocation of new replica locations. Repairing storage failures is slow and expensive. Losing a 250GB disk full of simulation data will cause the controlling servers to search for and replicate that much data over the network. The repair of this fault would be vain if the system suddenly came back up. If it does not come back, since the lost replicas may take a day or more to copy and restructure, other permanent fault combinations in the meantime could result in permanent data loss for one or more files. This puts a real cost on not just the storage space used by extra replicas, but also on the actions taken to respond to changes in the system, and stresses the need to order repairs effectively.

A recent project at the University of Notre Dame intends to store large quantities of biomolecular simulation data. This work intends to combine experimental and simulated scientific data on a dynamic, shared storage fabric. A variety of requirements for the storage system were quickly identified from the

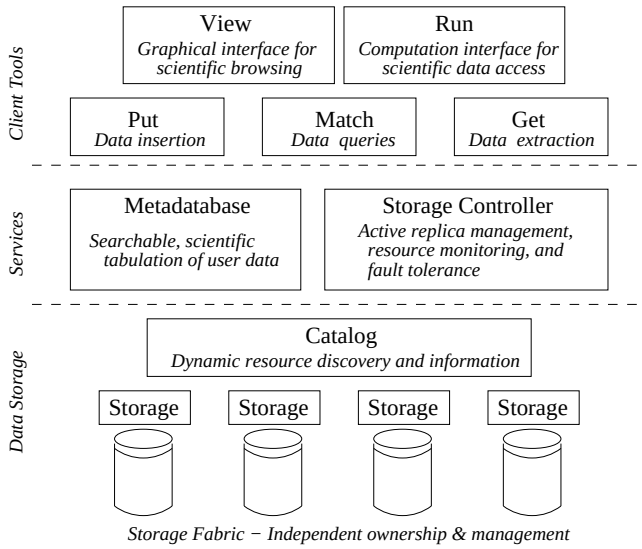

Fig. 1. GEMS Architecture

research domain, including data storage requirements relating to the high performance computational needs of researchers and scientific requirements that were to improve the utility of the storage system for exploring the valuable content of the data. A prototype was designed and quickly brought online to begin experimentation.

The system is designed to maximize the opportunities for scientific exploration by cataloging complete metadata for data in the system, allowing for automatically spawned simulations, data reuse, and the exploitation of relationships among data sets for research purposes. More abstractly, GEMS provides a framework for *analyzing*, *sharing*, and *archiving* important biomolecular data. The storage fabric for this system is a widely distributed uncontrolled network of servers that, individually, offer only best effort reliability. This provides an interesting observatory for the investigation of large scale fault tolerant systems built upon partially reliable servers.

The resulting system, named GEMS, for Grid Enabled Molecular Simulation, consists of client tools and a network of services that combine to provide the requested functionality to researchers in this actively evolving field. Figure 1 provides the skeleton of the system. This figure breaks GEMS into the client tools used to query the system, and the relationship among the GEMS services and underlying storage resources.

While GEMS provides a full set of tools, including a graphical browser and FTP-like get and put operations, the underlying API and protocol consist of three essential operations. *Match* is used to query the metadatabase with a variety of simulation parameters for existing files and replica locations. Original files and their simulation metadata are added to the system by reservation and committal. File operations in and out of the system are performed by Chirp

operations, over connections made directly from clients to storage servers. Data may be retrieved or created by running jobs with the *Run* tool that provides a virtual filesystem [Thain et al., 2005] to allow data access to general purpose programs.

We have previously discussed in [Wozniak et al., 2005] how GEMS manages high levels of disk loss, including the loss of half of the storage servers simultaneously. A method of reallocating the replication count of files was presented to reduce the probability of permanent data loss, resulting in a high degree of disk utilization control. In this paper, we intend to focus on improvements to the failure handling model to handle *subsequent massive failures*, such as the successive loss of whole storage clusters. In such a case, rapid dynamic replica creation would be necessary to prevent permanent data loss.

From the ground up, GEMS has been intended to respond to various faults that occur in large distributed systems. In this paper, we specifically focus on the effect of failures in the underlying storage servers to deliver requested data, or, even an acknowledgement. GEMS interprets these events and builds up an error handling system in response. Thus GEMS is fault-tolerant in the sense that it can continue to respond to queries and file requests despite the problematic storage layer upon which it operates. As discussed below, GEMS has knowledge of the metadata and data stored on its servers, and can *prioritize* response operations to reduce the probability of permanent loss of *important* data, and attempt to minimize the cost of replicating files unnecessarily.

In this paper, we compare the fault responses of GEMS to that of other systems, and demonstrate how GEMS meets the requirements of its research-based user pool. In the next section, we describe the GEMS model as it compares to other commonly used storage architectures and existing storage software systems. We then move into GEMS internal management techniques: Section 3 explains what constitutes a fault in the GEMS model and Section 4 explains the GEMS fault response system. Experimental results are provided in 5, and we offer some concluding remarks in Section 6.

2 GEMS Distributed Storage Model

GEMS has commonality with distributed file systems, databases, and peer-to-peer sharing systems. Users of these systems have different expectations when it comes to error states, and internally, faults are treated differently. Users of file systems and databases generally expect all-or-nothing responses. File system users expect that if one file on their machine is present and correct, then all will be be there. The locality assumption comes from the standard assumption that when a physical storage device fails, all of its data is perma-

nently lost, and that partial failures are handled by the operating system and not exposed to the user. This carries over into expectations of network file systems, which often attempt to emulate the behavior of a local file system.

While GEMS stores whole files and their directory information, it does not attempt to provide file system behavior. Files that are expected to be found in the same directory may be found on different hosts, destroying any assumptions about locality in the delivery of a file. This affects the design of clients to GEMS, which have utilize lists of replica locations. While GEMS does not expose internal faults to the user, the user should be aware of the actions that GEMS may take in the case of data loss, which include contacting various remote hosts.

NFS [Sandberg et al., 1985] is a standard point of comparison for file retrieval latency and API semantics for remote storage. GEMS is not designed to compete with a finely tuned cluster storage system, and since it is not a filesystem, it does not attempt to mimic any system's API semantics. GEMS does function in an RPC fashion from the client's perspective, and is stateless. On failure, GEMS does not block or disconnect, but continues to attempt to satisfy a user request, recreating data if possible. If permanent data loss has occurred, this will be reported and must be handled at a higher level.

AFS [Howard et al., 1988] provides remote file access, but provides some additional error handling functionality. Read-only replicas of stored data may be easily configured, and roll back to previous data is built in. However, AFS servers must maintain cache consistency for their clients. GEMS does not need to cache data because of its write-once characteristic, greatly reducing the server's responsibilities.

Relational database systems similarly are expected to provide all-or-nothing reponses. GEMS, however, may only be able to partially fulfill a query and will resort to partial delivery. GEMS does provide a metadatabase which may be queried, and normal database user expectations apply; when querying for output data files, the returned result is a fault-tolerant plan on how to retrieve the requested information from remote storage devices, and feed it into an external program.

GEMS can be compared to a peer-to-peer system because it allows users to query each others' data sets, and it allows users to pool storage resources. Peer-to-peer users have very low or non-existant expectations about reliability, for example, a file that is seen once could quickly vanish forever. However, GEMS is a write-once system, so that data that is published to the system cannot be modified, which is quite different from the expectations of most storage users, but not so surprising in a setting in which data is "published" to other users. GEMS data is much more reliable than an *ad hoc* peer-to-peer network,

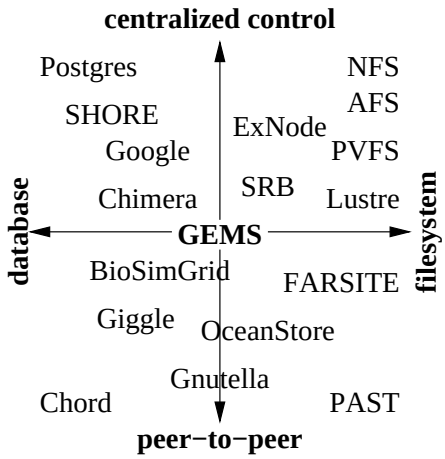

Fig. 2. Related Work

because it dynamically creates replicas as needed to keep data alive.

The most obvious point of comparison for a redundant data storage system is a hardware RAID [Patterson et al., 1988], or a network-based RAID, such as Zebra [Hartman and Ousterhout, 1993]. In a typical hardware RAID setting, faults may be detected as blocks are read from the device, where in GEMS, errors are actively probed by a server component. Zebra detects faults over the network, which is a techically difficult observation to make. GEMS and RAID both benefit from hot-pluggable hardware. As devices are added, both systems can discover and begin utilizing new storage. When a new server is added to GEMS, it may immediately begin receiving data from a client, just as some RAID types may immediately use a new disk. Removing a disk from either may be performed while the system is running. GEMS offers some additional benefits in that it has knowledge of the importance and replication status of the stored data, so upon replica loss, replication does not necessarily immediately consume network bandwidth for all lost data.

As diagrammed in Figure 2, a coordinate system may be considered to compare various systems by their properties. GEMS is placed as an intermediary between ideal filesystems and databases, and between totally centralized systems and pure peer-to-peer systems. GEMS is comparable to scientific databases for simulation like BioSimGrid [Nga et al., 2006] and Chimera [Foster et al., 2002]. Chord [Stoica et al., 2001], Gnutella [Ripeanu and Foster, 2002], and PAST are fully decentralized systems, contrasted with centralized systems like Postgres [Stonebraker and Rowe, 1986] and NFS [Sandberg et al., 1985] tend toward centralized control, all of which represent extremes on the diagram. Databases like SHORE [Carey et al., 1994] and Google [Brin and Page, 1998] represent moderately centralized databases, likewise, PVFS [Carns et al., 2000] and Lustre [Systems, 2002] represent moderately centralized filesystems. Giggle [Chervenak et al., 2002], OceanStore [Sit et al., 2006], and FARSITE [Adya et al., 2002] represent replica systems, object systems, and filesystems

that tend toward peer-to-peer characteristics.

With these summarized comparisons, we can describe GEMS in the context of modern data grids [Finkelstein et al., 2004]. The GEMS architecture combines a few commonly observed architectures. The storage sites may be thought of as peer-to-peer clusters that combine to contribute to a complete storage system, each offering file space and receiving replicas. Since the system is actively probed for data loss by a variety of server components, it can accurately be described as an agent controlled system. The active components can reside on various servers, which lends greatly to fault tolerance and scalability, but creates the increased possibility of partial failure in the system and requires a shared critical resource for inter-component communication.

This hybrid architecture allows our fault handling policy to utilize the metadata archive to make intelligent choices concerning replication and repair. It also meets other requirements of the system, such as the ability for contributing storage servers to revoke storage with limited impact, and simplifies the security aspects of such a shared system.

Additional recent work has developed the data grid and its implementation in various systems. From a management perspective, GEMS constitutes a centralized, dynamic replica management system as defined previously [Lamehamedi and Szymanski, 2006]. GEMS application users gain performance and replica locality in a manner similar to that offered by European Data Grid [Cameron et al., 2005] projects such as PROuST [Ciriello et al., 2006], or a variety of other data location aware grid methodologies [Tang et al., 2006, Ranganathan and Foster, 2003, Venugopal et al., 2004].

3 Fault Management and Maintenance

The GEMS framework is subject to many of the same error modes that plague its cousin grid distributed systems. Sample sources of faults include: an errant user, the desktop hardware, the OS, the TCP connection, the local switch, numerous routers, the storage server software and its hardware, etc. Even though the apparent severity of one of the mentioned failures may differ by orders of magnitude (the loss of one node due to hard drive failure versus the loss of an entire DNS domain due to switch failure), the GEMS design embraces the spectrum of failure with one unified resolution policy. *Observed faults are viewed not as pre-emptive exceptions, but events on par with typical maintenance operations.*

To give this concept substance we introduce the concept of a Problem. A Problem is an object in GEMS which can be assigned a priority based on its

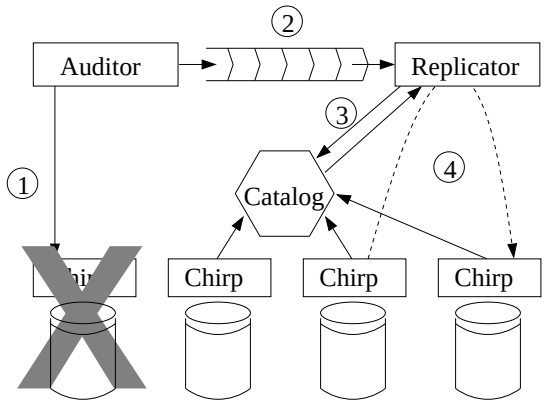

Fig. 3. GEMS Prioritized Repair

severity and queued for resolution. Problems are generated through Auditors which continously audit the system state with respect to the metadata, physical reality, and user requests as reflected in the metadata. On the failure side of the Problem continuum, one example would be the loss of a local switch providing access to ten nodes, say, *Cluster A*. The Auditor would recognize that the metadata pointing to *Cluster A* is now inconsistent and for each unavailable file a Problem will be instantiated with all of the pertinent information necessary for calculation of a Problem priority and resolution. On the maintenance side of the Problem continuum, consider a client who submits a record of ten files into the database with a requested replication of 5 each. The replica will make sure the first copy is submitted to stable storage by the client tool but the remaining requested copies will be handled through the instantiation of Problems as the metadata shows an insufficient number of replications with respect to the users request. Hence, Problems can be subtyped to provide a useful abstraction for error handling.

With this unified response to both failures and maintainance tasks we can exploit varied risk/cost targets through specification of those metrics which comprise the prioritization function.

4 Prioritization of Failure Handling Events

The prioritization of Problems is central to the priority queue fault tolerance model. In this work we do not argue for a specific priority function nor discuss the suitability of those parameters which are included in the function. We do however present our prototype's implementation of the priority calculation and mention select parameters from our priority function in order to substantiate a functional implementation. The flexible prioritization function allows the specific user community to tune the system to their risk/cost demands.

As shown in Figure 3, the Auditor iteratively probes Chirp servers to verify their contents against the GEMS metadatabase. When any of a variety of faults are detected (1), the relevant error information is encapsulated into a Problem object and enqueued (2). When the Replicator pops the queue, it determines a response, usually by creating an additional replica. Existing copies of the file in question are located by consulting the metadatabase and a suitable host for the new replica is found by reading the Chirp catalog (3). The file is then copied over (4).

Each Problem object contains a set of members which are necessary for the Problem's resolution and which also serve as the parameters for the priority calculations. From the set of priority members a priority is calculated internally by the Problem on instantiation. Of key interest is the ability for the Problem to recalculate its own priority. For example, the priority queue could trigger all Problems in the queue to recalculate their priority (stale Problem member data is addressed in the following section).

A good demonstration of the priority recalculation utility leads us into the discussion of priority function parameters. Classical job starvation concerns would prompt us to include "time in queue" as a parameter with varies in proporition to an increased priority. Problems involving a host failure which has only been recognized for a short period are not handled immediately, which gives the host system a chance to recover, eliminating an unnecessary file transfer. A second parameter of importance for our scientific user base is file type. Whereas an output file may be automatically regenerated because the metadata contains enough information to derive the output files, the input configuration files are irreplacable without human intervention. To again reduce the probablity of permanent data loss we increase priority in with respect to the number of remaining replications. Additional parameters such as file size allow us the ability to fine tune the prioritization for improved response to massive failure scenarios. An important observation in biomolecular simulation data sets is that input files tend to be small, and output files are large. This allows us to aggressively replicate the input files while being more cavalier about output files, which saves storage space and bandwidth.

4.1 Stale Problems

System state data is inherently stale in a distributed framework, and is expensive to obtain, as discussed in the introduction. Therefore we expect that the state which was recognized when the Problem was instantiated may not reflect the system state when it is handled. In response our failure handler must validate that the member data is still current. If the Problem data remained in sync with the current metadata it is resolved as normal, however, if the data is

no longer current the priority status is re-evaluated. If the priority is reduced, as in the case of a returning storage server, a new Problem will be created and queued, and if the Problem is eliminated, the Problem is discarded. If the priority of the new Problem exeeeds or matches that of the stale problem it is, of course, resolved immediately.

4.2 Failed Failure Responses

One challenge in the field of fault tolerance has been the determination of when to abort attempts at repair if the repair operation itself continues to fail. Although the GEMS response to the failure of one system is chiefly the replication of a remaining copy of the file on a second machine to a new destination on a third machine, one can imagine that by the time the Problem is being handled the second machine with the remaining copy has also gone down. The typical answer would be to retry at regular intervals with an eventual timeout. The GEMS model improves upon this by allowing a Problem to count its history of failed repairs, and adjust its priority accordingly. Problems are re-queued when repair fails, so that other repairs may be attempted. Since the age of a Problem is also recorded, extremely old Problems can signal for human intervention.

4.3 Starvation and Priority Inversion

Queue structures prompt the user to investigate the fate of those jobs relegated to the back. The GEMS prioritization function as we have implemented it includes consideration of the time spent in the queue to prevent starvation. The priority queue enables preferential response unavailable to sequential queues but does not guarantee against priority inversion even for a large number of Problem handlers running in parallel. Our model allows for Problems to reprioritized *while the repair is in progress*, and the Replicator to be interrupted and forced to pop the queue again, by a third agent that probes for priority inversion. Thrashing is avoided by prioritizing a Problem currently being repaired with respect to the size of the data yet to be replicated, and the fact that it is in progress. Since our priority function includes a term which is inversely proportional to the square root of the size, copies in progress are unlikely to be interrupted except in the case of real emergencies, as defined by the tuned coefficients.

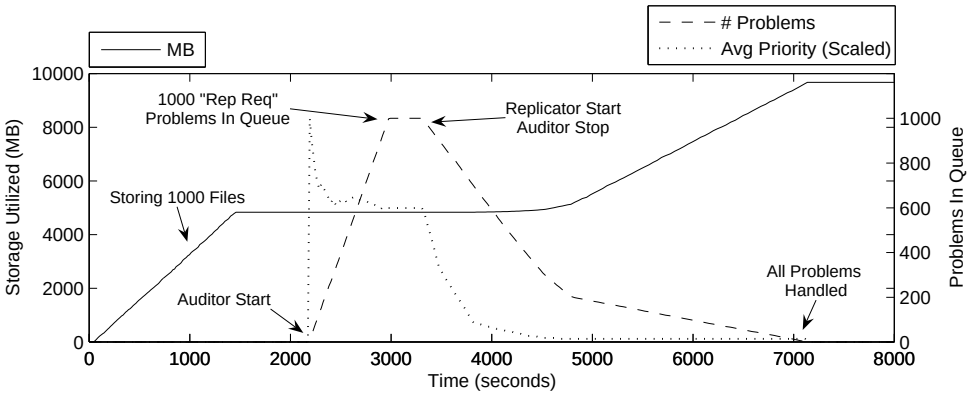

Fig. 4. Queue Performance: Priority Dominated by File Size

4.4 Scaled Response on par with Problem Queue Size

For a sufficient number of Auditor components detecting faults, it is reasonable to propose that the size of the problem queue is indicative of the number of failure handlers required. Furthermore the average priority of Problems in the queue provides insight into the severity of the situation and should influence the number of failure handlers. Although the experimental results presented in the work reflect a single failure handler, the authors plan to implement scaled instantiation of failure handling components on other GEMS servers with respect to priority queue metrics.

5 Experimental Results

The authors have developed a functional GEMS prototype as reported in [Wozniak et al., 2005]. Specific to this work we implemented the Problem Queue and report experimental data in demonstration of the error recovery capability.

We demonstrate that the new functionality improves upon the previous functionality, which is similar to functionality obtained in a system with limited access to simulation metadata. Such a data-agnostic system could only repair errors in the order observed, one at a time, and could not respond dynamically to a rapidly changing storage fabric. Our prototype can emulate this behavior if we make all the priorities equivalent, and perform in-order response.

Our example shows that the prototype can respond to faults in the order specified by the priority function. In this case, small files have priority over

large files, which is typical in a biomolecular simulation environment where the input files and configuration files are irreplaceable, and the output files represent derived data and are only required to speed the retrieval time.

The underlying experimental setup consisted of the client machine, the GEMS server, and an underlying storage network of various heterogeneous Linux machines from the Notre Dame campus network running the Chirp file server. The client machine, a 900MHz AMD Linux system, controlled the experiment by communicating with the GEMS server, a dual-processor 2.8GHz AMD Linux machine, via the GEMS clients.

As demonstrated in Figure 4, we start by disabling the GEMS storage management components, the Auditor, Replicator, and GarbageCollector. The simulation data sets, about 3 GB total, are added to the system, so each file has one copy on some Chirp server. The Auditor is then turned on and replica shortages are detected: the files need to be replicated up to the requested level. The graphic shows the average priority of problems, which are scored by, in this case, the size of the file and the replication count. The Replicator is turned on, and the average priority quickly drops as high priority problems are quickly handled by replicating these small files on other Chirp hosts. The average priority slowly levels off as larger, low priority output files are copied over the network, increasing the disk usage level.

6 Conclusion

Researchers that produce and use biomolecular simulation data have needs, both computational and scientific, for a storage system that reliably catalogs and stores simulation data and metadata. Scientifically, the system must be centered around the metadata and provide a virtualized data framework, which makes output data recreateable and new explorations more systematic. Computationally, the metadatabase allows the storage policy to act with substantially more intelligence than existing distributed or replica systems. In this paper, we explained the necessary observations about the data in question and showed that it can be prioritized by value and risk in a variety of tunable ways. We provided our Problem model, which encapsulates information about any of a variety of observed situations in a sortable context, which facilitates appropriate response with respect to other conditions in the system.

Because the risk/cost parameters are difficult to obtain in advance, we will continue experimentation to improve the system performance with respect to its most important metric: permanent loss of input files. In the meantime, we provide some observed behavior that shows that a first-try prioritization scheme is much better than no prioritization, especially important on volatile

network of unreliable storage servers.

References

A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. Symposium on Operating Systems Design and Implementation*, 2002.

S. Brin and L. Page. The anatomy of a large scale hypertextual search engine. *Computer Networks and ISDN Systems*, 30(1–7), 1998.

D. Cameron, J. Casey, L. Guy, P. Kunszt, S. Lemaitre, G. McCance, H. Stockinger, K. Stockinger, G. Andronico, W. Bell, I. Ben-Akiva, D. Bosio, R. Chytracek, A. Domenici, F. Donno, W. Hoschek, E. Laure, L. Lucio, P. Millar, L. Salconi, B. Segal, and M. Silander. Replica management in the european datagrid project. *J. Grid Computing*, 2(4), 2005.

M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *Proc. SIGMOD*, 1994.

P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Annual Linux Showcase and Conference*, 2000.

A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, and M. Ripeanu. Giggle: A framework for constructing scalable replica location services. In *Proc. Supercomputing*, 2002.

G. Cirielloa, M. Comina, and C. Guerra. Algorithmic re-structuring and data replication for protein structure comparison on a grid. *Future Generation Computing Systems*, 23(3), 2006.

A. Finkelstein, C. Gryce, and J. Lewis-Bowen. Relating requirements and architectures: A study of data-Grids. *J. Grid Computing*, 2(3), 2004.

I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proc. Scientific and Statistical Database Management*, 2002.

J. Hartman and J. Ousterhout. The Zebra striped network file system. In *Proc. Symposium on Operating System Principles*, 1993.

J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *Transactions on Computer Systems*, 6(1), 1988.

H. Lamehamedi and B. K. Szymanski. Decentralized data management framework for data grids. *Future Generation Computer Systems*, 23(1), 2006.

M. H. Nga, S. Johnston, B. Wuc, S. E. Murdock, K. Tai, H. Fangohr, S. J. Cox, J. W. Essex, M. S. Sansom, and P. Jeffreys. Biosimgrid: Grid-enabled biomolecular simulation data storage and analysis. *Future Generation Computer Systems*, 22(6), 2006.

D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. Management of Data*, 1988.

K. Ranganathan and I. Foster. Simulation studies of computation and data scheduling algorithms for data grids. *J. Grid Computing*, 1(1), 2003.

M. Ripeanu and I. Foster. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. In *Proc. Workshop on Peer-to-Peer Systems*, 2002.

R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network File System. In *Proc. USENIX*, 1985.

E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz. Proactive replication for data durability. In *Proc. Workshop on Peer-to-Peer Systems*, 2006.

I. Stoica, R. Morris, D. Karger, M. Kaashock, and H. Balakrishman. Chord: A scalable peer-to-peer lookup protocol for internet applications. In *Proc. SIGCOMM*, 2001.

M. Stonebraker and L. A. Rowe. The design of Postgres. In *Proc. SIGMOD*, 1986.

C. F. Systems. Lustre: A scalable, high performance file system. White paper, November 2002.

M. Tang, B.-S. Lee, X. Tang, and C.-K. Yeo. The impact of data replication on job scheduling performance in the data grid. *Future Generation Computer Systems*, 22(3), 2006.

D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *Proc. Supercomputing*, 2005.

S. Venugopal, R. Buyya, and K. Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Computing Surveys*, 38(1), 2006.

S. Venugopal, R. Buyya, and L. Winton. A grid service broker for scheduling distributed data-oriented applications on global grids. In *Proc. Middleware for Grid Computing*, 2004.

J. M. Wozniak, P. Brenner, D. Thain, A. Striegel, and J. A. Izaguirre. Generosity and gluttony in GEMS: Grid-Enabled Molecular Simulation. In *Proc. High Performance Distributed Computing*, 2005.