Applying Feedback Control to a Replica Management System

Justin M. Wozniak, P. Brenner, D. Thain, A. Striegel, J. A. Izaguirre

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

***Abstract*— Many modern storage systems used for large-scale scientific systems are multiple use, independently administrated clusters or grids. A common technique to gain storage reliability over a long period of time is the creation of data replicas on multiple servers, but in the presence of server failures, ongoing corrective action must be taken to prevent the loss of high value and low value data. Such a system is difficult to control, and replica management is typically handled in an *ad hoc* manner. In this work, we claim that repairing prioritized faults is a scheduling problem, founded on the need to minimize a risk-based error function, E. Citing experiments on a prototype replica system for molecular simulations, we apply concepts from control system theory to analyze and handle the application of corrective action.**

I. Introduction

Modern high performance scientific computing has evolved into a community exercise in which large groups of users share computers and storage resources. The conglomeration of middleware and standards used to manage such a system is called the grid [4]. A grid computing system allows the user to access a much larger number of sites for computation as well as participate in large scale, shared storage systems.

Large, shared storage systems create new problems. As users increasingly depend on remote systems for storage, the underlying systems become less trusted and reliable for a variety of reasons. A storage provider could revoke storage by filling the storage device, evicting users, or simply turning the machine off. Additionally, the scale of the number of hardware components involved implies that individual components will fail regularly.

To achieve fault-tolerance and data preservation over long periods of time, replicas of each datum stored in the system are created and spread across large numbers of devices. The creation of replicas amplifies the size and scope of the data management problem. Additional replicas demand additional storage space. Appropriate metadata regarding the status of the replicas must be constantly maintained to keep the system intact. When data is to be retrieved, existing replicas must be located, and access to the data must be granted to the user. In a writable system, changes to the data in one replica must be propagated to all copies in a consistent way. Additionally, the system is dynamic. Appropriate response actions must be taken if a file can no longer be accessed. Analogously, action must be taken if a whole storage device or file server becomes unresponsive.

In this paper, we describe our solution to the replica management problem in the application area of molecular simulation, and provide observations that apply to replica systems in general. We discuss existing tools for replica management in Section II, and sketch our solution, Grid-Enabled Molecular Simulation (GEMS) [10], in Section III. In Section IV we provide our model for replica systems in general, and show how it can be used to analyze the actions of a replica management system. We offer some conclusions in Section V.

II. Related Work

A variety of systems to support replica management on distributed computer systems have been developed. A community leader in the promotion of standards, protocols, and software for grid computing is the Globus Alliance [1]. The open-source Globus Toolkit includes the Replica Location Service (RLS) [3]. The RLS provides the ability to map logical file names to physical file locations, thus providing a building block for distributed replica storage. Systems built upon the RLS must manage metadata externally.

Another important replica system is the Storage Resource Broker, (SRB), which provides the user with an abstraction layer over a variety of underlying storage systems. SRB provides an "all-hosts" replication technique and a user-controlled technique, and manages the metadata catalog in a database [7].

A storage system designed for the application area of molecular dynamics is BioSimGrid [8]. Centering on a simulator-independent scientific database, BioSimGrid provides tools to perform analysis on its libraries of simulation data. The software architecture combines a standard database with an underlying SRB storage system.

III. Case Study: Grid-Enabled Molecular Simulation

An extremely active area of modern research in high performance computing centers around applications in molecular dynamics [6]. Molecular dynamics (MD) is the numerical simulation of bonded atoms and the forces they exert on each other over time. Typical solvated single protein systems on

the order of 10,000 atoms require weeks of simulation time on distributed sytems to compute motion on the nanosecond time scale. Storage for sufficient post analysis of this relatively small system is on the order of gigabytes (10^9 bytes) and biochemists are aggressively working to achieve simulations results on micro and milisecond timescales, rapidly pushing the storage requirements for individual simulations into the terabyte (10^{12} bytes) range. Storage repositories for simulations would require petabyte (10^{15} bytes) storage capacity just to facilitate current simulation results.

The combination of MD simulators and modern grid middleware provides the researcher with a powerful tool for performing large numbers of experiments, multiplying the work done and storage required. In fact, the amount of storage required by one user may be much more than is commonly available on one hard disk, and even a conventional network of storage servers may not be enough. Users must look elsewhere to find storage.

Additionally, molecular dynamics is an interdisciplinary endeavor which often combines large numbers of researchers from geographically and organizationally distant places. The data from a simulation may be reused to the benefit of all researchers in a given project. This means that users are often interested in sharing data with others which implies that the storage fabric itself is shared.

The GEMS system has been designed to meet these goals: to increase the amount of storage available to users, provide a searchable catalog of metadata, and allow for data sharing and reuse. The system consists of a three-level architecture: a toolset of client programs, including a GUI, a central metadatabase and storage management system, and the distributed storage servers. Users access the system through the client tools in a variety of ways. They can add data sets to the system, using *GEMSput*. The system may be queried for existing data using *GEMSmatch*. New data runs are simplified by using *GEMSrun*, which allows the user to spawn new simulations by connecting the user to a compute system called APST [2], and managing the metadata in a GEMS-compatible way. Users are expected to typically just use the *GEMSrpr* tool, which performs a high-level result production request, utilizing the three lower-level clients.

The storage layer used is the Chirp [9] personal file server, which runs on each storage machine volunteered to the system. The Chirp servers register with a central catalog so that resource information may be automatically discovered. Chirp provides file services with UNIX semantics, a variety of authentication methods, and simplified, user-level administration.

GEMS is an active replication system. Over time, the server processes actively probe for problems and determine a response, typically creating additional replicas or garbage collecting storage for later use.

The active maintenance in GEMS consists of three components, the Auditor, Replicator, and Garbage Collector. The Auditor uses the metadatabase as a guide, and contacts the storage servers to determine whether the actual system is consistent with the information in the database. If a fault is

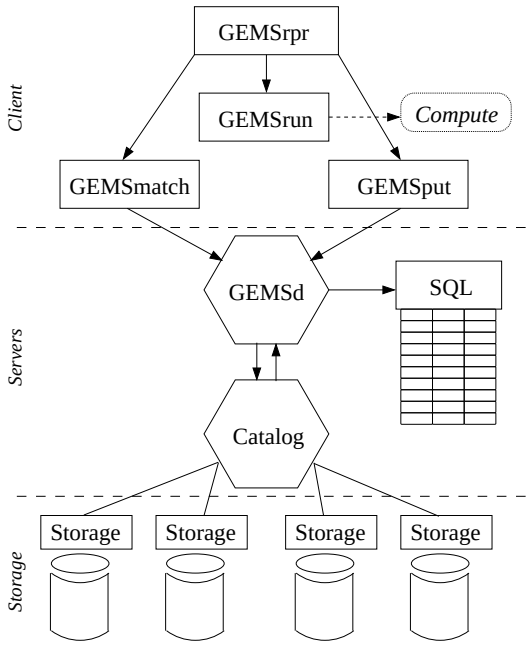

Fig. 1. GEMS Toolset Framework.

detected, relevant information is compiled into a Problem object, ranked with a priority value, and enqueued in a *Priority Queue*. The Replicator monitors the Priority Queue, removes the Problem object representing the highest ranked fault, and determines a response. In the case of a typical server failure, the lost file must be identified by consulting the metadatabase. Existing copies of this file are looked up, free space is located on a Chirp storage server, and the file is copied to restore the replica count to its desired level. If files are over-replicated, or otherwise inserted into the GEMS storage space inconsistently with the metadatabase, they will be detected by the Garbage Collector component and deleted to free space.

The maintenance process is shown in Figure 2. The *Observation* step in the control loop represents the active probing of the Auditor component for faults. Corrective action is shown as the *Correction* signal from the controller.

IV. A Model for the Analysis of a Replica Management System

In this section, we discuss some principles to allow the use of control system theory to analyze our replica management system. We begin with basic definitions and symbols, and then describe some basic tools from control system theory that prove useful when analyzing the GEMS system.

A. Definitions

Fundamental to the analysis of a replica system is the number of replicas of a given file. The system contains N files, where each file f_i has x_i replicas, $f_{i1}...f_{ix_i}$. Each replica must be stored on a server c, which means that for each f_{ij} in existence, $\exists k : f_{ij} \in c_k$. We loosely say $f_i \in c_k$ if $\exists j : f_{ij} \in c_k$. The replica count is constantly changed as

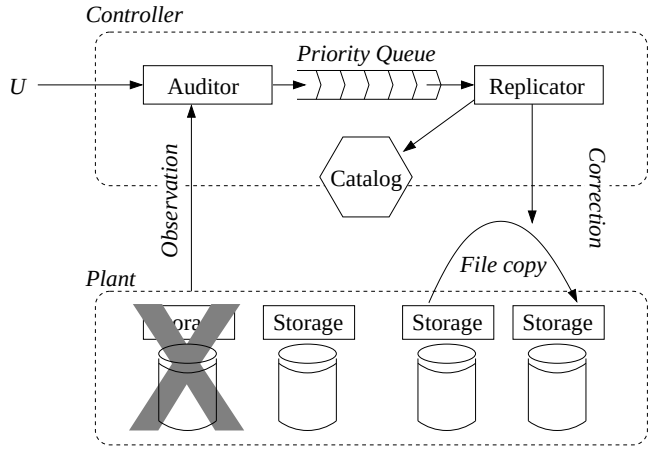

Fig. 2. Replica Control Loop.

files are lost due to storage loss, or as files are replicated, so we have $x_i(t)$ representing the number of replicas for f_i at time t. If $x_i = 0$, then f_i is permanently lost. Each file has a size in bytes, s_i. The total storage consumed by the system can be expressed as:

$$S(t) = \sum_{i=1}^{N} s_i x_i(t). \tag{1}$$

The number of storage servers and the amount of storage offered is also constantly changing over time. Of the M storage servers, d_j represents the amount of disk space offered by server c_j. Thus physical storage limit of a server represents a constraint on the files that may be stored there:

$$\forall c_j, \sum_{i:f_i \in c_j} s_i < d_j. \tag{2}$$

The amount of available storage C in the system at a given time t may be expressed as:

$$C(t) = \sum_{j=1}^{M} d_j(t). \tag{3}$$

A clear consequence of (2) is that $C \geq S$.

At the time of data insertion into the system, the user may specify how many replicas are requested for a given file, which we call u_i. If $x_i < u_i$, then the user's desired replication level is not being met, and replication should occur. The total storage requested by all users is:

$$U = \sum_{i=1}^{N} s_i u_i. \tag{4}$$

We consider U to be the reference signal to the GEMS system; the GEMS system is constructed to keep the underlying storage in line with the requested replica level.

B. System Response Analysis

We begin this discussion with an example: an experimental use of the system over a short period of time, as shown in Figure 3.

This experiment proceeded over a period of 5 hours. A GEMS installation was configured, and access was granted to Chirp storage servers running on 19 machines. The system was configured such that the amount of apparently available storage for GEMS was near 350 GB. The given diagram plots the amount of storage apparently available as observed internally by GEMS, and the amount of storage apparently consumed as observed by GEMS. Both observations are significantly delayed behind real time.

For the first 1.5 hours, data was inserted into the system using GEMSput. The input data consisted of 50 files, each 337.6 MB in size, each replicated 10 times. At hour 1.5, the system contained 500 files. Shortly after data insertion, a fault was induced on one of the storage servers, causing it to lose all GEMS data. This means that those replicas were permanently lost, but that the amount of storage available was not reduced. The effect is almost imperceptible on this plot. At hour 2, faults were induced on 4 servers, causing a noticeable bump in the amount of storage apparently consumed. At hour 3, faults were induced on 7 servers, causing an even more significant bump.

This type of diagram may be compared to a diagram describing the response of a system to an impulse. In this simple experiment, since all the files are equivalent, the input signal can be described as $U = 500 \times 337.6\text{MB}$, requiring the system to maintain 500 files of 337.6 MB each. Deviation from U may be measured over time to determine how poorly the system is able to respond to the forced change in state. A variety of error functions may be developed to quantify the performance of the system.

A simple metric is simply the storage used by the system. GEMS should not leave free space unused when additional file replicas are requested. For t in the runtime of the experiment, the error function in (5) simply measures how well GEMS is filling the available space with data.

$$E = \int_t \min(C(t), U(t)) - S(t)\ dt. \tag{5}$$

Equation (5) offers no information about what is filling

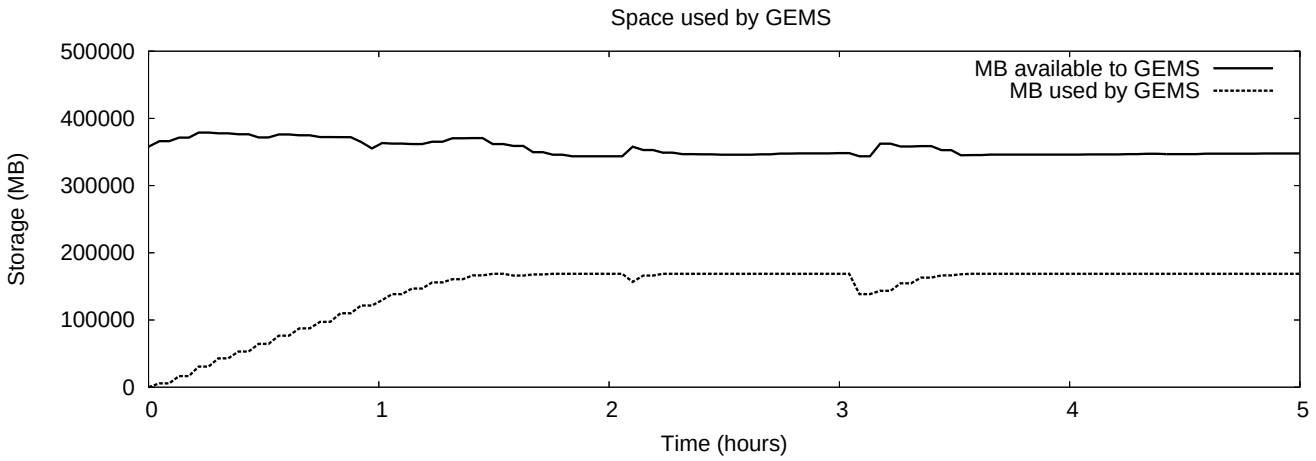

Fig. 3. System Reponse to Induced Server Faults.

that space, whether it is high priority data, or even whether some files are over-replicated. If the system is full, i.e. $U \geq C$, this equation is a equivalent to:

$$E = \int_t \sum_{i=1}^{N} (u_i - x_i) s_i \; dt. \tag{6}$$

Since the system should not be rewarded for over-replication, we have:

$$E = \int_t \sum_{i=1}^{N} (\max(u_i - x_i), 0) s_i \; dt. \tag{7}$$

For many purposes, including fairness, file size is almost irrelevant to the worth of data. In many cases, if a single file is lost, of any size, the amount of work that the researcher would have to do to diagnose the problem and rescript the simulation run is similar. Strictly in terms of per-file replication counts, one could measure unweighted replica shortages as:

$$E = \int_t \sum_{i=1}^{N} (\max(u_i - x_i), 0) \; dt. \tag{8}$$

A related measurement is the number of files that have been completely lost due to the loss of all of their replicas. We measure this as:

$$Z(t) = (\text{The count of files permanently lost.}) \tag{9}$$

C. Optimal System Response

All files are not created equal. Most users of storage systems can specify which files are more important than others. GEMS makes this easy, and provides a few ways of specifying data worth to the system. As discussed in Section III, GEMS maintains a great deal of metadata about the datasets it stores which can be used.

The performance of a replica management system should be measured in terms of what the users require from the system. This is partially captured in the difference $u_i - x_i$, but this does not capture the value of the data.

A simple observation from the experience of running simulations, developed more fully in [5], is that input files are more valuable than output files. This is especially true in the GEMS environment, which is designed to promote the sharing of input files: it is convenient to have pre-computed output files, but if lost they may always be recomputed by combining the input files and the execution information from the GEMS metadatabase. Additionally, other factors may weight the value of data files.

Data value would not be important if GEMS had unlimited ability to respond to faults, but this is not the case. The typical response to a fault, as described in our experiment, is to create an additional replica. The ability to create additional replicas is limited by several factors, including the availability of source servers, destination servers, and the network. GEMS limits the stress on servers by ensuring that the Replication process never makes use of a given server for more than one task at a time, i.e., a server is either transmitting a single file, receiving a single file, or idle. GEMS does not currently explicitly limit its consumption of network resources.

The result of these observations and constraints is that faults in a large system such as GEMS must be prioritized: they can not all be handled or repaired at once, and they do not all represent a threat to data of the same value. This line of thinking led to the creation of the Priority Queue in the GEMS controller, between the Auditor and Replicator in Figure 2.

The architecture necessary is simple enough, but the priority assignment scheme has not been dealt with. Given a certain set of faults, what is the appropriate action to take?

If we assume that we have an error function E that is correct, then we simply must minimize E. At each opportunity, the system must take the action that will likely result in the minimum value of E. Since future values of the cost function are affected by unknown, unpredictable events such as server failures, we must make standard assumptions about

Problem Object

Field	Type
RequestedReplicas	`integer`
CurrentReplicas	`integer`
Duration	`date`
Size	`integer`
InputFile	`boolean`
FileMetadata	`object`
Priority	`integer`

Fig. 4. Problem Object in GEMS

the future status of the system. For example, we can assume that all servers are equally likely to fail, regardless of the content that is stored on them. This implies that we may minimize E by locally minimizing the cost function. If the cost function E is easy to understand, this is easy to do.

Our cost function is simply the sum of a set of values that result from observed faults. Each fault is prioritized in some way. So the set of actions that we may take is made up of the set of corrective actions we may take to repair an observed fault, and minimizing the cost function is equivalent to repairing the highest priority fault first.

D. Determining a Priority System

The remaining problem is to find a method to pick an appropriate E. This is now equivalent to the problem of prioritizing faults. The priority of a fault is then some function applied to the available information about that fault.

In a typical replica management system, there are several available statistics about a fault. We have the number of requested replicas from the user, the number of replicas intended to be allocated by the system, the number of observed existing replicas on storage servers, the time elapsed since the fault was detected, and the size of the affected file. In the GEMS system, we also have information about whether the file was an input file to a simulation. GEMS represents each observed fault as a Problem object, with fields summarized in Figure 4.

Each Problem observed by the Auditor represents a file that is below its requested replica count and thus is in need of additional replication. The *RequestedReplicas* field indicates how many replicas of this file were requested by the user, and indicates indirectly how valuable the user feels this file is. The *CurrentReplicas* field indicates how many replicas currently exist in the system, which is a volatile observation. The system stamps each Problem with the time of observation, stored under *Duration*. The size of the file is stored as *Size*. File size may affect the resulting response in a variety of ways, for example, the system may choose to delay replicating a very large file to allow hundreds or thousands of smaller file copies to complete first: performance which may be desirable under many of the error definitions given above. The *InputFile* field is *true* if the user has indicated that the given file is an input file to a simulation. *FileMetadata* is a compound field that provides the filename and replica locations, and is needed to perform the repair. Based upon all of this information, the *Priority* field can then be used to determine which Problem to process next, highest *Priority* first.

In GEMS, the priority function (P) is related to the above error functions but is also influenced by more general notions of fairness, as well as known observations about the underlying storage system. For example, it is known that many storage servers that are unavailable will eventually come back, so a delay (D) component is introduced, and the priority is based around the duration in minutes (M) of the Problem, scaled by the risk of total file loss (K). We adopt the convention that Problems with $P \leq 0$ are left in the queue. Currently:

$$P = MK - D \tag{10}$$

Future work in developing GEMS will bring the priority computation into relationship with the control-theoretic error functions describe above.

V. Conclusion

Replica control systems have complex dynamics that must be controlled carefully to minimize the risk of the loss of high and low valued data. There is a need to quantify the state of the system and its current level of risk in a cost function. The resulting error function for the system may be then be used to control the system in a systematic way, as opposed to an *ad hoc* manner.

In this paper, we demonstrated some simple functions that are useful when analyzing a certain replica control system, GEMS. We then showed how this error function can be used to make decisions when faults are observed in this system.

References

[1] The Globus Alliance. `http://www.globus.org` .

[2] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS parameter sweep template: User-level middleware for the grid. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2000.

[3] A. L. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf. Performance and scalability of a replica location service. In *Proceedings of the International Symposium on High Performance Distributed Computing (HPDC-13)*, 2004.

[4] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid. *Intl. J. Supercomputer Applications*, 15, 2001.

[5] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, 2002.

[6] T. Schlick. *Molecular Modeling and Simulation - An Interdisciplinary Guide.* Springer-Verlag, New York, NY, 2002.

[7] G. Singh, S. Bharati, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A metadata catalog service for data intensive applications. In *Proceedings of Supercomputing*, 2003.

[8] K. Tai, S. Murdock, B. Wu, M. Ng, S. Johnston, H. Fanghor, S. J. Cox, P. Jeffreys, J. W. Essex, and M. S. P. Sansom. BioSimGrid: towards a worldwide repository for biomolecular simulations. *Org. Biomol. Chem.*, 2, 2004.

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

[10] J. M. Wozniak, P. Brenner, D. Thain, A. Striegel, and J. A. Izaguirre. Generosity and gluttony in GEMS: Grid-Enabled Molecular Simulation. In *Proceedings of the International Symposium on High Performance Distributed Computing*, 2005.