Faults in Grids: Why are they so bad and What can be done about it?

Raissa Medeiros, Walfredo Cirne, Francisco Brasileiro, Jacques Sauvé
{raissa,walfredo,fubica,jacques}@dsc.ufcg.edu.br
Universidade Federal de Campina Grande – Paraíba - Brazil

Abstract

Computational Grids have the potential to become the main execution platform for high performance and distributed applications. However, such systems are extremely complex and prone to failures. In this paper, we present a survey with the grid community on which several people shared their actual experience regarding fault treatment. The survey reveals that, nowadays, users have to be highly involved in diagnosing failures, that most failures are due to configuration problems (a hint of the area's immaturity), and that solutions for dealing with failures are mainly application-dependent. Going further, we identify two main reasons for this state of affairs. First, grid components that provide high-level abstractions when working, do expose all gory details when broken. Since there are no appropriate mechanisms to deal with the complexity exposed (configuration, middleware, hardware and software issues), users need to be deeply involved in the diagnosis and correction of failures, when, in fact, all they want is to run their applications. One needs a way to coordinate different support teams working at the grids different levels of abstraction. Second, fault tolerance schemes today implemented on grids tolerate only crash failures. Since grids are prone to more complex failures, such those caused by heisenbugs, one needs to tolerate tougher failures. Our hope is that the very heterogeneity, that makes a grid a complex environment, can help in the creation of diverse software replicas, a strategy that can tolerate more complex failures.

1 Introduction

The use of computational grids as a platform to execute parallel applications is a promising research area. The possibility to allocate an enormous amount of resources to a parallel application (thousands of machines connected through the Internet) and to make it with lower cost than traditional alternatives (based in parallel supercomputers) is one of the main attractive in grid computing.

In fact, grids have the potential to reach unprecedented levels of parallelism. Such levels of parallelism can improve the performance of existing applications, and raises the possibility to execute entirely new applications, with huge computation and storage requirements. On the other hand, grid characteristics, as high heterogeneity, complexity and distribution – traversing multiple administrative domains – create many new technical challenges, which need to be addressed.

In particular, grids are more prone to failures than traditional computing platforms. In a grid environment there are potentially thousands of resources, services and applications that need to interact in order to make possible the use of the grid as an execution platform. Since these elements are extremely heterogeneous, there are many failure possibilities, including not only independent failures of each element, but also those resulting from interactions between them (for example, a task may fail because the browser version in a specific grid node is not compatible with the Java version available). Moreover, machines may be disconnected from the grid due to machine failures, network partitions, or process abortion in remote machines to prioritize local computation. Such situations cause non-availability of the processing service, characterizing failure scenarios.

Dealing with these complex failure scenarios is challenging. Detecting that something is wrong is not so difficult (in general, symptoms are quickly identified), but difficulties arise to identify the root cause of the problem, i.e., to *diagnose a failure* in a very complex and heterogeneous environment such as a computational grid.

The first barrier is to understand what is really happening and the problem here seems to be a *cognitive* one. It is often possible to obtain logs and information about the resources that compose the grid. However, in order to make sense of this information, one would have

to know what *should* be happening. In a grid context, this means to understand the functioning of the many different technologies that compose it. When failures occur and the transparency provided by the middleware is compromised, the user needs to drill down to lower level of abstractions in order to locate and diagnose failures. This requires understanding many different technologies in terms of middleware, operating systems and hardware. It is just too much for any single human being!

Note that some solutions for grid monitoring have been proposed [1] [2] [3] [4] [5] [7] [8]. They are certainly useful, since they allow for failures detection and also ease the collection of data describing the failure. However, they do not provide mechanisms for failure diagnosis and correction, so grid users are unhappy because they need to be too much involved in these highly complex tasks. Moreover, fault-tolerant solutions (such as [6] [14] [17]) address only crash failure semantics for both hardware and software components. Software faults with more malign failure semantics, such as those caused by heisenbugs [9], are not covered by them.

Consequently, dealing with failures in grids is current a serious problem for grid users. No wonder that, in a survey we conducted, grid users said that they are highly involved in diagnosing failures, that most failures are due to configuration problems (a hint of the area's immaturity), and that solutions for dealing with failures are mainly application-dependent.

In this paper, we describe the *status quo* of failures in grids. In Section 2 we present a survey that exposes the difficulties highlighted above. The aim of this survey was to capture the actual experience, regarding fault treatment, of those who have been using grids as a computational environment. In Section 3, we show why the available solutions are not sufficient to treat faults in grid environments in an effective manner. Further, in Section 4, we point research directions that could be taken in order to facilitate the grid fault treatment and to provide software fault tolerance in a grid environment. Section 5 concludes the paper with our final remarks.

2 The *Status Quo* of Failures in Grids

In order to identify the *status quo* of failures in grids, we have consulted grid users spread throughout the world through the multiple-choice questions below.

1. What are the more frequent kinds of failures you face on Grids?
2. What are the mechanisms used for detecting and/or correcting and/or tolerating faults?
3. What are the greatest problems you encounter when you need to recover from a failure scenario?
4. To what degree is the user involved during the failure recovery process?
5. What are the greatest users´ complaints?
6. Are there mechanisms for application debugging in your grid environment?

A full version of the questionnaire is available at http://www.dsc.ufcg.edu.br/~raissa/survey/form.html. The questionnaire was sent on 11 April 2003 to several grid discussion lists, such as:

- users@gridengine.sunsource.net
- centurion-sysadmin@cs.virginia.edu
- wp11@datagrid.cnr.it
- users@cactuscode.org
- agupta@phys.ufl.edu
- vaziri@nas.nasa.gov
- condor-admin@cs.wisc.edu
- grads-users@isi.edu
- support@entropia.com
- mygrid-l@dsc.ufcg.edu.br
- developer-discuss@globus.org
- discuss@globus.org
- gridcpr-wg@gridforum.org
- grid@cnpq.br

Answers were received via email and Web. On 25 April 2003, we had 22 responses. It is interesting to note that a similar survey (i.e. a self-selected survey conducted on-line) with users of parallel supercomputers resulted in 214 responses [18], an order of magnitude higher than our survey. Furthermore, many respondents have demonstrated a high level of interest about the results of our research, signing their hope for better ways to deal with failures in grids. These facts highlight the infancy of grid computing and that better fault treatment is a key to bring grids to maturity.

2.1 The Survey

Kinds of Failures

The main kinds of failures (see Figure 1) are related to the environment configuration. Almost 76% of the responses have pointed this out. According to some

people surveyed, the lack of control over grid resources is the main source of configuration failures. Following this, we have middleware failures with 48%, application failures with 43% and finally hardware failures with 34%. Note that, in the majority of the responses, more than one kind of failure was chosen.

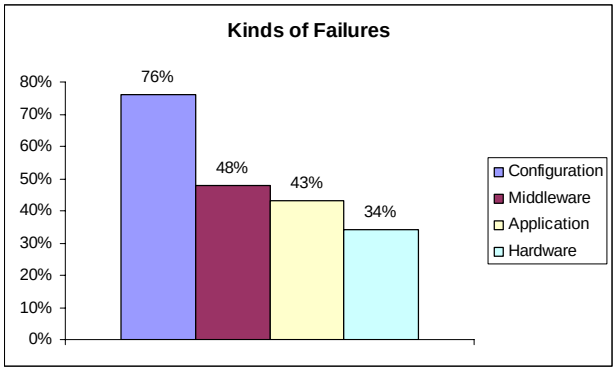

Figure 1: Kinds of failures

Fault Treatment Mechanisms

In addition to ad-hoc mechanisms – based on users' complaints and log files analysis – grid users have used automatic ways to deal with failures on their systems (see Figure 2). Nevertheless, 57% of them are application-dependent. Even when monitoring systems are used (29% of the cases) they are proprietary ones (in fact, standards such as GMA [1] and ReGS [8] are very new specifications and have few implementations). Checkpointing is used in 29% of the systems and fault-tolerant scheduling in 19%. In some cases, different mechanisms are combined.

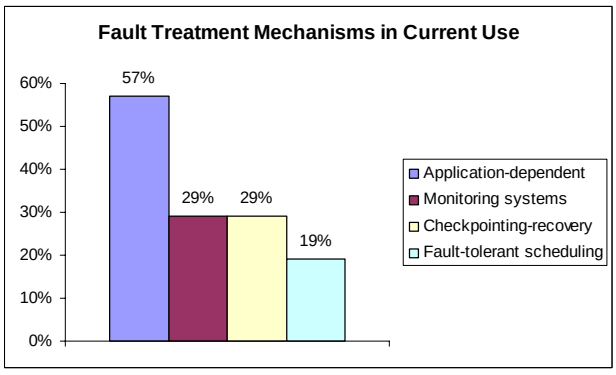

Figure 2: Fault Treatment Mechanisms in Current Use

In case of checkpointing-recovery and fault-tolerant scheduling, they are only able to deal with crash failure semantics for both hardware and software components. Software faults with more malign failure semantics – such as timing or omission ones, which are even more difficult to deal with - are not covered by them.

The Greatest Problems for Recovering from a Failure

The greatest problem is to diagnose the failure, i.e. to identify its root cause. About 71% of the responses have pointed this out (see Figure 3). The difficulty to implement the application-dependent failure recovery behavior is present in 48% of the cases (the user does not know what to do to recover from a failure), and to gain authorization to correct the faulty component is a problem in 14% of cases.

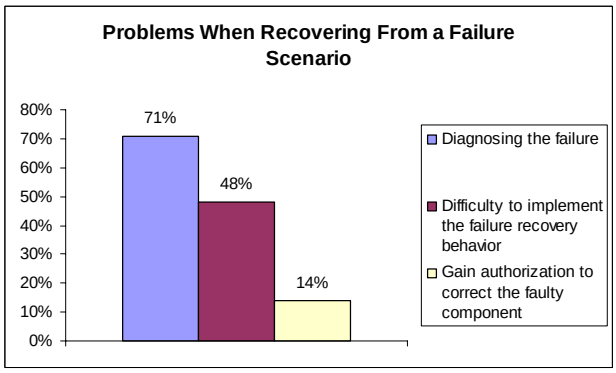

Figure 3: Problems When Recovering From a Failure Scenario

Other problems such as ensuring that failures do not result in orphaned jobs on remote systems (i.e. they get cleaned up in a reasonable time), cleaning up corrupted cache files without losing lots of work in progress, and getting access to preserved state when checkpointing-recovery is used (e.g. checkpoint files may be inaccessible or totally lost) were also highlighted.

Degree of User Involvement

As the above results suggest, the user needs to be highly involved during the failure recovery process (see Figure 4). About 58% of them need to define exactly what should be done when failures occur (which is not an easy task). 29% of them are somewhat involved - e.g. the user can specify at submission time if he/she should be notified when serious errors happen or if the system should attempt to recover as best as it can, resulting in orphaned jobs etc. Only 13% of the users are involved in a low degree and can rely on the mechanisms provided by the system.

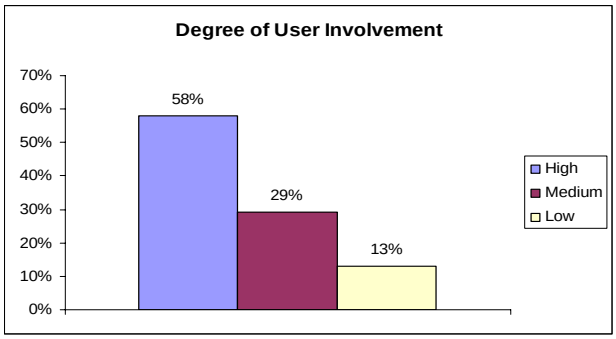

Figure 4: Degree of User Involvement

The Greatest Users´ Complaints

When we asked about the users´ complaints, the main result is related to the complexity of the failure treatment abstractions/mechanisms (71% - see Figure 5). Once more, the users are concerned with the ability to recover from failures, more than the failure occurrence rate (33%) or the time to recover from them (10%).

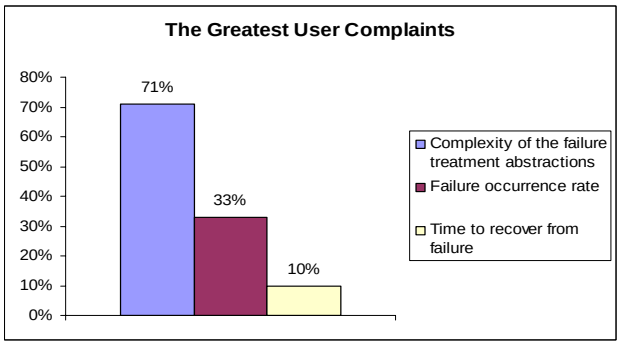

Figure 5: The Greatest Users´ Complaints

Application Debugging

The following result highlights a clear open issue in grid computing: grid users do not have appropriate mechanisms for application debugging (see Figure 6). Less than 5% (just one response) have good mechanisms that allow them to influence the application execution (e.g. change a variable value); 14% have "passive mechanisms" that only allow them watching the application execution; 19% have mechanisms that do not show them a grid-wide vision of their application (i.e. the mechanism scope is limited to a single resource that comprise the grid); and 62% of the grid users have no available application debugging mechanism.

The lack of debugging mechanisms almost suggest that grid developers believe that applications have no bugs and will operate correctly despite of grid complexity and heterogeneity. Unfortunately, the reality is quite different.

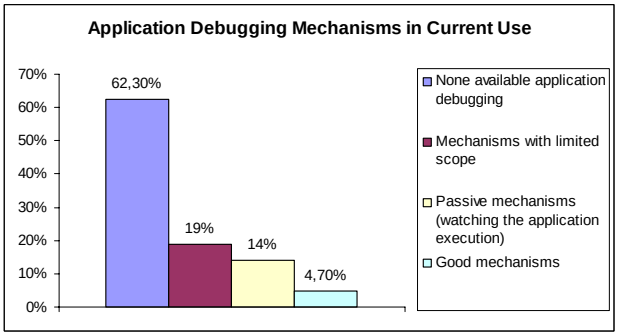

Figure 6: Application Debugging Mechanisms in Current Use

2.2 Survey Lessons

From the responses above, we can infer that grid users are unhappy, since failures are not rare and they cannot rely on appropriate failure treatment abstractions. They are using application-dependent solutions, so they need to be too much involved in the time-consuming and complex task of dealing with failures. The main source of failures is related to configuration issues and failure diagnosis is the main problem.

This scenario is a result of the following fact: grid application developers use abstractions provided by grid middleware to simplify the development of application software for such a complex environment that is a grid. Similarly, grid middleware developers use abstractions provided by operating system to ease their jobs. This is an excellent way to deal with complexity and heterogeneity, except when things go wrong. When a software component malfunctions, it typically affects the components that use it. This propagates up to the user, who sees the failure. Then, in order to solve the problem, one has to drill-down through abstraction layers to find the original failure. The problem is that, when everything works, one has to know only *what* a software component does, but when things break, one has also to know *how* the component works. Although not exclusive of grids, this characterization is a much bigger problem in grids than in traditional systems. This is because grids are much more complex and heterogeneous, encompassing a much greater number of technologies than traditional computing systems. In a grid, one can discover a failure in a grid processor about what he/she could never know its hardware platform model has existed. Thus he/she know nothing about it. He/she does not know how it

should work. He/she does not know where its logs are. Thus, solving the problem is a very difficult task.

Therefore, there is a huge cognitive barrier between the failure detection and the failure diagnosis. Most of the time the logs are available, indicating a problem, but who reads them can not interpret them. Consequently, grid fault treatment depends on intensive user collaboration, including not only system administrators but also application developers. In this way, the focus of application developer is lost when he/she would probably like to concentrate on application functionality, rather than diagnosing middleware or configuration failures. The available solutions are unable to overcome this cognitive problem as we will see on the next section.

3 Existing Solutions

There are solutions available for grid fault treatment. However, most of them were designed with performance analysis in mind [1] [2] [3] [4] [7] [8] and they basically provide an infrastructure for grid monitoring. Of course, the information collected on the grid resources and/or applications may be used for several purposes, including failure detection and diagnosis. However, these solutions do not solve the cognitive problem described above.

The GMA (Grid Monitoring Architecture) [1], for instance, is an open standard being developed by the *Global Grid Forum Performance Working Group* for grid monitoring. As such, it can be used as a template solution through which we can describe grid monitoring solutions in general.

Its architecture consists of three types of components, shown in Figure 7. The *directory service* supports information publication and discovery. The *producer* makes management information available. The *consumer* receives management information and processes it.

Typically, the information exchanged between the components is described as events, a data collection with a specific structure defined by an event schema. Events are always sent directly from a producer to a consumer. The data used to produce events may be gathered from several sources and any of the following may be data sources: hardware or software sensors that collect real-time measurements (such as CPU load, memory usage etc), databases, monitoring systems (such as JAMM [3]) and applications with their specific events.

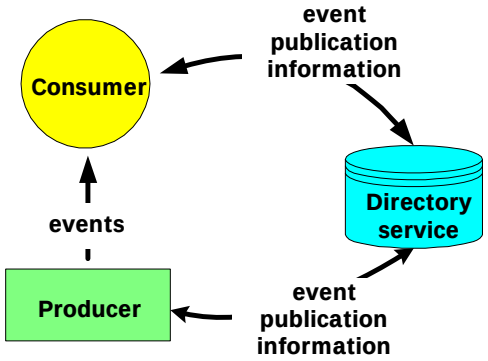

Figure 7: A general monitoring architecture [1]

Consumers, in turn, may have different functionalities, using the received information for several purposes. Some consumers examples are: a real-time monitor, which provides information for real-time analysis; an archiver, which stores information for future use; an event correlator, which makes decisions based on events gathered from different sources; a process manager; which restarts services once process failures occur. In any case, the consumer behavior is defined by the application.

Besides producers and consumers, it is possible to design new components, called intermediaries, which implement both interfaces simultaneously to provide specialized services. For instance, an intermediary can collect events from several producers, produce new data derived from the received events and make this information available to other consumers. The Reporting Grid Services (ReGS) system [8] specifies two kinds of intermediaries for OGSA [15] application monitoring: an intermediary for filtering events and another for logging.

As we can notice, grid monitoring solutions are concerned with the gathering of information across grid nodes. However, the problem does not seem to be gathering data, but having the knowledge to use them. Since there is no available mechanism to help diagnosing the failure once it is detected, a consumer that performs failure diagnosis and recover must know what the events should look like, identify the events that do not match with the expected pattern, and devise a suitable way to tackle this mismatch. All the knowledge encapsulated into the consumer is defined by the application.

There are also solutions focusing on fault tolerance, rather than grid monitoring. Such solutions strive to make the application run correctly even in the presence of crash failures. Solutions such as GALLOP [6] and WQR [14], for instance, use task replication to provide fault tolerance. GALLOP replicates SPMD (single-program-multiple-data) applications in different sites within the virtual organization, while WQR is an efficient fault-tolerant scheduler for *bag-of-tasks* applications. If a task fails, the user is not aware of it and the solution reschedules the task automatically. Certainly, in order to prevent undesirable side-effects due to replica execution, these solutions allow for committing tasks results only in the end of the execution.

Checkpoint-recovery has also been used. Although this mechanism is difficult to do for parallel jobs with tasks spread across multiple processors where messages may be in transit [6], systems such as Legion [19] and Condor [20] provide fault tolerance through it. In Legion system, checkpoint-recovery is provided in the application level; in Condor, it is embedded into the system level.

Some of the survey respondents have been using both checkpoint-recovery and fault-tolerant scheduling solutions (see Figure 2). In all cases, however, they deal only with crash failure semantics for both hardware and software components. They do not deal with software faults or faults with more malign failure semantics, despite grids being even more prone to these kind of failures, as is detailed in Section 4.2.

4 What Can Be Done About It?

It is necessary to look for solutions that allow managing the complexity involved in grid fault treatment in an efficient manner. Application developers or users should not be involved on diagnosis and correction of middleware or configuration failures. We see improvement needed in both (i) failure diagnosis and correction, and (ii) fault tolerance.

4.1 Failure Diagnosis and Correction

In order to solve the cognitive problem that no one is going to know all details of a grid when failures occur, it should be possible to define different hierarchical levels of abstraction. At each hierarchical level, appropriate personal (e.g. application developer, middleware administrator and system support staff) should be responsible for dealing with faults. In this way, if a failure is detected on a higher layer, but its root cause is at a lower one, the corresponding staff should be activated to solve the problem. The challenge is to identify the right levels for this hand-on, allowing collaborative drilling-down in a controlled and effective manner. Ideally, the hand-off points should be narrow interfaces.

Besides, it may be necessary to define mechanisms to coordinate the interaction between the different groups to fix the problems. Once these mechanisms are available, debugging tools could take advantage of them. A possible mechanism is an automated test of a given service. Automated tests are key for enabling the staff solving a problem at layer n to determine whether the problem is their own or is at layer n - 1, without understanding how layer n - 1 works. Note that, although components are exhaustive tested before going into production, the ability to run tests in production is very useful. It allows for finding configuration errors and even bugs that were not detected in the developers' environment. Additionally, automated tests ease not only problem hand-on. After using the tests for the lower layer and concluding that the problem is at their own layer, support staff can use the tests for their own layer to expedite the problem isolation.

4.2 Fault Tolerance

Besides the issue of failure diagnosis and correction, there is also another interesting question to be considered in terms of fault treatment. It is important to investigate how to provide broader fault tolerance in grids, since grid software (middleware and applications) is complex and, as all complex software, prone to failures that are more malign than crashes, such as timing or omission ones. Fault tolerance mechanisms such as replication and checkpointing-recovery have been used in grid systems. However, as highlighted above, they are only able to deal with crash failure semantics.

Special care should be taken with heisenbugs, i.e. software bugs that lead to intermittent failures whose conditions of activation occur rarely or are not easily reproducible [10]. Heisenbugs cause a class of software failures that typically surface in situations where there are boundaries between various software components [11], and thus they are likely to appear in grids. Note that, by their very nature, heisenbugs result in intermittent failures

that are extremely difficult to identify through testing. This is particularly preoccupying because we have just seen that automated tests may play a very important role in failure diagnosis and correction in grids, but they can take no effect when facing with heisenbugs.

Software fault tolerance is provided by software diversity [12] [13] [14]. Diversity can be introduced in software systems by constructing diverse replicas that solve the same problem in different ways (different algorithms, different programming languages etc). The idea is to make different replicas to fail independently and so to avoid a specific failure to compromises the whole processing.

Since grids are extremely heterogeneous, one might be able to take advantage of this diversity to provide software fault tolerance through software diversity. In grids, if on one hand the compilers, operating systems and hardware heterogeneity can increase the system complexity, on the other hand it can potentially facilitate the construction of diverse software replicas, thus increasing software reliability. In particular, it is interesting to investigate how to introduce software diversity *automatically*, rather than involving different and independent groups of programmers to develop each replica. In this sense, randomized compilation techniques [12] may be a starting point. Furthermore, replicas could be scheduled and executed in different grid nodes where different hardware architectures or programming languages could be available.

5 Conclusions

In this paper we described the *status quo* of failures in grids. A survey we conducted with grid users showed that they are not pleased with the current state of affairs. The survey revealed that users have to be highly involved in diagnosing failures, that most failures are due to configuration problems (a hint of the area's immaturity), and that solutions for dealing with failures are mainly application-dependent.

We identified two basic problems in grid fault management. First, existing solutions for failure diagnosis and correction mainly address information collection. However, while in principle one has to know only *what* software component does, when such a component breaks, one has also to know *how* the component works. Unfortunately, there are too many different components in a grid. It is not reasonable to expect for a single human to master all details of a grid. We propose the definition of specific hand-on points for different support teams to cooperate in diagnosing and correcting grid problems. In this way, application, middleware and resource problems can be handled in a coordinated manner. Such a cooperative effort would be much helped by automated tests.

Second, fault tolerance schemes today implemented on grids tolerate only crash failures. Since grids are prone to more complex failures, such as heisenbugs, one needs to tolerate tougher failures. Our hope is that the very heterogeneity that makes a grid a complex environment can help in the creation of diverse software replicas, a strategy that can tolerate more complex failures.

Acknowledgments

We would like to thank Paulo Roisemberg and Daniel Paranhos for a number of useful comments and criticisms. This research was supported by grants from Hewlett Packard/Brazil, CNPq/Brazil and CAPES/Brazil.

References

[1] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany. *A grid Monitoring Architecture*. Working Document, January 2002, http://www-didc.lbl.gov/GGF-PERF/GMA-WG /papers/GWD-GP-16-2.pdf

[2] W. Smith. *A Framework for Control and Observation in Distributed Environments*. NASA Advanced Supercomputing Division, NASA Ames Research Center, Moffett Field, CA, NAS-01-006, June 2001.

[3] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, and M. Thompson. *A Monitoring Sensor Management System for Grid Environments*. Proceedings of the IEEE High Performance Distributed Computing Conference (HPDC-9), August 2000.

[4] A. Waheed, W. Smith, J. George, and J. Yan. *An Infrastructure for Monitoring and Management in Computational Grids*. In Proceedings of the 2000 Conference on Languages, Compilers and Runtime Systems, 2000.

[5] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. Laszewski. *A Fault Detection Service for Wide Area Distributed Computations*. Proc. of the 7th IEEE Symp. On High Performance Distributed Computing, 1998, pp. 268-278.

[6] J. Weissman. *Fault Tolerant Computing on the Grid: What are My Options?* Technical Report, University of Texas at San Antonio, 1998.

[7] M. Baker, and G. Smith. *GridRM: A resource Monitoring Architecture for the Grid*. The Distributed Systems Group, University of Postsmouth UK, June 2002.

[8] Y. Aridor, D. Lorenz, B. Rochwerger, B. Horn, and H. Salem. *Reporting Grid Services (ReGS) Specification*. IBM Haifa Research Lab, draft-ggf-ogsa-regs-0.3.1, January 2003.

[9] J. Gray. *Why do Computers Stop and What Can Be Done About it?* Tandem Computers, Technical Report 85.7, PN 87614, June 1985.

[10] K. Vaydianathan and K. S. Trivedi. *Extended Classification of Software Faults based on Aging*. Dept. of ECE, Duke University, Durham, USA, 2001.

[11] S. Forrest, A. Somayahi and D. H. Ackley. *Building Diverse Computer Systems*. In Proceedings of The 6th Workshop on Hot Topics in Operating Systems, IEEE Computer Society Press, Los Alamitos, CA, pp. 67-72, 1997.

[12] A. Avizienis. *The N-Version Approach to Fault-Tolerant Software*. In IEEE Transactions in Software Engineering SE-11(12), pp. 1491-1501, 1985.

[13] B. Randell. *System Structure for Fault Tolerance*. In Yeh R T (Ed) Current Trends in Programming Methodology (Vol 1), Prentice-Hall, Englewood Cliffs, NJ, 1977.

[14] D. Paranhos, W. Cirne and F. Brasileiro. *Trading Information for Cycles: Using Replication to Schedule Bag of Tasks Applications on Computational Grids*. In Proceedings of the Euro-Par 2003: International Conference on Parallel and Distributed Computing, August 2003.

[15] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. *Grid Service Specification*. Draft 3, Global Grid Forum, July 2002. http://www.globus.org/research/ papers/ gsspec.pdf.

[16] ETTK home page. http://www.alphaworks.ibm. com/tech/ettk

[17] A. Tuong and A. Grimshaw. *Using Reflection for Incorporating Fault-Tolerance Techniques into Distributed Applications*. University of Virginia, Department of Computer Science, September 1999.

[18] W. Cirne and F. Berman. *A Model for Moldable Supercomputer Jobs*. Proc. IPDPS 2001: International Parallel and Distributed Processing Symposium, April 2001.

[19] A. S. Grimshaw, A. Ferrari, F. Knabe and M. Humphrey. *Wide-Area Computing: Resource Sharing on a Large Scale*. IEEE Computer, May 1999.

[20] M. Litzkow, M. Livny, and M. Mutka. *Condor – A Hunter of Idle Workstations*. In Proceedings of the 8th International Conference of Distributed Computing Systems, pp. 104-111, June 1988.