A Flexible Fault-Tolerance Mechanism for the Integrade Grid Middleware

Stanley Araujo de Sousa
and Francisco José da Silva e Silva
Universidade Federal do Maranhão
Av Portugueses, S/N, Bacanga, São Luis, MA, Brasil
Email: stanleyaraujo@oi.com.br, fssilva@deinf.ufma.br

Rafael Fernandes Lopes
Centro Federal de Educação Tecnológica
Departamento Acadêmico de Informática
Av. Getúlio Vargas, 04, Monte Castelo, São Luis, MA, Brasil
Email: rafaelf@cefet-ma.br

***Abstract*— Computer grids have attracted great attention of both academic and enterprise communities, becoming an attractive alternative for the execution of applications that demand huge computational power, allowing the integration of computational resources spread through different administrative domains. The dynamic nature of the grid infrastructure, its high scalability, and great heterogeneity exacerbates the likelihood of errors occurrence, imposing fault tolerance as a major requirement for grid middlewares.**

This paper describes a flexible fault-tolerance mechanism implemented on Integrade grid middleware that allows the customization of several failure handling parameters and the combination of different failure handling techniques. This paper also presents several experiments that measure the benefits of our approach, considering several different execution environments scenarios.

I. Introduction

A computer grid comprises a hardware and software infrastructure that allows integration and sharing of distributed resources, such as software, data and peripherals, inside and among institutions. This computational infrastructure has attracted great attention of academic and enterprise communities, becoming an attractive alternative for execution of applications that demand huge computational power, and allowing the integration of computational resources spread through different administrative domains.

Computational grids have been used to solve problems in varied areas of scientific, enterprise, and industrial activities, such as: computational biology, image processing for medical diagnosis, weather forecast, high energy physics, marketing simulations, and oil prospection. Grid computing has empower the conception of a new generation of applications that allow combining computations, experiments, observations, and data got in real time. The phenomena modeled by these applications require diverse software components whose compositions and interactions are extremely dynamic. Moreover, the grid infrastructure is also heterogeneous and dynamic, aggregating a great amount of computation and communication resources, databases and, sometimes, sensors and specific peripherals. The dynamism can be observed in terms of high variation in resource availability, node instability, and workload variations in nodes and network links.

The dynamic nature of the grid infrastructure, its high scalability, and great heterogeneity has turn impracticable its configuration, maintenance and recovery in case of failures solely by human beings. Several recent research projects [10], [9] have recognize the necessity of providing a greater autonomy to grid systems, which comprises one of the greatest challenges for the new generation of this kind of middleware. The term *autonomic computing* has been used to denote a system that exhibits the following four functional properties [7]:

- **Self-Protection**: the system should be capable of detecting and protecting its resources from both internal and external attacks, maintaining its overall security and integrity;
- **Self-Optimization**: the system should be able to detect performance degradation and intelligently perform self-optimization actions;
- **Self-Healing**: the system must be aware of potential problems and should have the ability to reconfigure itself in order to continue to function smoothly;
- **Self-Configuration**: the system must have the ability to dynamically adjust its resources based on its state and the state of its execution environment.

The AutoGrid project, currently being developed at the Federal University of Maranhão, main goal is the development of a robust and self-managing autonomic grid system. The AutoGrid project uses the Integrade grid middleware [6] as the foundation for its implementation, incorporating autonomic mechanisms to its infrastructure in order to make its configuration and administration independent from human intervention. Our research focuses on adding to Integrade three autonomic properties: self-healing, self-optimization, and self-configuration.

This paper presents our initial effort toward AutoGrid self-healing infrastructure: the development of a flexible failure handling mechanism. The generic, dynamic, and heterogeneous nature of a grid environment requires a failure handling mechanism that supports multiple fault tolerance techniques, since each technique performs better (i.e. causes a smaller overhead over the application execution time) for a given environment condition. We argue that the decision about which technique should be applied in each situation must be taken by the grid middleware in an autonomic and automatic way, based on the environment status and previous experiences.

This paper is organized as follows: Section II presents main issues concerning fault tolerance mechanisms for grid environments. Section III describes the architecture of the Integrade middleware. Section IV describes the design and implementation of Integrade flexible failure handling mechanism. Section V presents the performance evaluation and a comparative analysis among the failure handling techniques implemented with a discussion about which technique is better in each situation. Section VI shows some related works, while Section VII presents our conclusions and describes the next steps of this work.

II. Fault Tolerance in Grid Environments

Computer grid environments are highly prone to failures due to several facts, such as [12]: (a) Grids systems are composed of a wide range of services, software, and hardware components, which need to interact with one another. System failures can result not only from an error on a single component but also from the interaction between components; (b) Grid environments are extremely dynamic, with components joining and leaving the system all time; (c) The likelihood of errors occurrence is exacerbated by the fact that many grid applications will perform long tasks that may require several days of computation.

To provide the necessary fault tolerance functionalities for grid environments, several services must be available, such as:

Failure detection: grid nodes and applications must be constantly monitored by a failure detection service. Two approaches can be used: In the *push model*, grid components periodically send heartbeat messages to a failure detector, announcing that they are alive. In this approach, the monitor suspects the failure of a component in the system after a certain time interval. However, if there is a large number of monitored components its heartbeat messages can flood the network. In contrast, in the *pull model* the failure detector sends liveness requests (*"are you alive?"* messages) periodically to grid components. In this case, the load on the network is reduced and depends on the number of liveness requests sent by the monitor. However, the monitor may not suspect or detect the failure of a component until after it sends it a liveness request;

Application Failure handling: diverse failure handling strategies can be applied in grid environments in order to ensure the continuity of applications executions. The main application failure handling techniques adopted in grid environments are:

- *Retrying*: when an application execution fails, it is restarted from scratch. This is the simplest failure handling technique but its main drawback is the loss of computation time in case of failure;
- *Replication*: the same application is submitted for execution a number of times, generating various application replicas. All replicas are active and perform the same code with the same input parameters at different nodes. This technique can tolerate only the occurrence of up to $n-1$ failures (only crash failures), where n is the total number of replicas;
- *Checkpointing*: periodically saves the process state in a stable storage during the failure free execution time. Upon a failure, the process restarts from one of its saved states, thereby reducing the amount of lost computation. Each of the saved states is called a checkpoint. The use of this technique usually imposes some overhead over the execution time, caused by the periodic saving of the execution state.

Stable storage: execution states that will allow to recover the pre-failure state of applications must be saved in a data repository that can survive eventual failures of grid nodes. A stable storage can be implemented using a centralized or distributed approach.

A. Flexible fault tolerance mechanism

Grid environments have a heterogeneous nature in respect to its tasks (e.g., long running tasks, mission critical tasks, transactional tasks, etc) and its execution environment (e.g., highly reliable execution environments, unreliable execution environments). This heterogeneity leads to the necessity of a flexible failure handling mechanism that supports multiple fault tolerance techniques, allowing each task to select an appropriate fault tolerance technique among alternatives depending on its characteristics and the estimated reliability of its underlying execution environment. For example, if a grid computing resource on which a task is running has a long downtime[1], the task may prefer the "retrying on another available grid resource" strategy to either the "retrying on the same resource" or "restarting with checkpointing on the same resource" strategies. In Hwang et al. [8], one can found a comparative analysis based on simulations among the four main failure handling techniques: *retrying*, *checkpointing*, *replication* and *replication with checkpoint*. The necessity for providing flexibility to the grid failure handling mechanism comprises a major requirement of our work on the Integrade grid middleware.

III. Integrade Overview

The Integrade project [6] is a multi-university effort to build a novel grid computing middleware infrastructure to leverage the idle computing power of personal workstations for the execution of computationally-intensive parallel applications. The basic architectural unit of an Integrade grid is the cluster, a collection of machines usually connected by a local network. Clusters can be organized in a hierarchy, allowing to encompass a large number of machines. Each cluster contains a *Cluster Manager* node that executes Integrade components responsible for managing the cluster computing resources and for inter-cluster communication. Other cluster nodes are called *Workstations*, which export part of its resources to Grid users. They can be shared or dedicated machines. The main components of Integrade architecture are:

[1]Average time between the task failure and it is up again.

- **Application Submission and Control Tool** (ASCT): a graphical user interface that allows users to submit applications and control their execution;
- **Application Repository** (AR): stores the code of applications that can be executed on the grid;
- **Local Resource Manager** (LRM): a component that runs in each cluster node, collecting information about the state of resources such as memory, CPU, disk, and network usage. It is also responsible for instantiating and executing applications scheduled to the node;
- **Global Resource Manager** (GRM): manages the cluster resources by receiving notifications of resource usage from the LRMs (through an information update protocol), and runs the scheduler that allocates tasks to nodes based on resources availability;
- **Execution Manager** (EM): maintains information about each application submission, such as its state, executing node, input and output parameters, submission and conclusion timestamps. It also coordinates the application recovery process, in case of failures.

A. *Application execution protocol*

Figure 1 shows the interactions among Integrade components in order to execute applications. The user requests an application execution through the ASCT interface. The application must have been previously registered in the application repository. The user can, optionally, assign some requirements for the execution, such as the platform in which the application was compiled and the minimum amount of memory necessary for its execution. ASCT forwards the request to the GRM (1), which executes its scheduling algorithm in order to select an available cluster node. GRM notifies EM that the application execution was scheduled (2) and forwards the submission data to the LRM of the selected node (3).

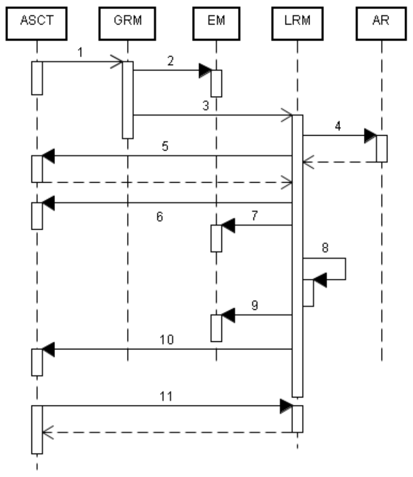

Fig. 1. Integrade application execution

The scheduled LRM downloads the (previously registered) application binary from the AR (4), requests the application input files (if any) to the ASCT (5), notifying the acceptance of its request (6). Before starting the application execution, the LRM notifies EM about the application initialization (7), and starts to monitor the execution progress (8), waiting for its conclusion. At this time, the LRM notifies the EM (9) and ASCT (10) the end of the application execution. The ASCT can now download the results (output files) of the executed application (11).

IV. Integrade Fault Tolerance Mechanism

The original Integrade fault tolerance mechanism was completely based on application level checkpointing. Application-level checkpointing consists on instrumenting the application code to periodically save its state, thus allowing recovery after a fail-stop failure. Integrade provides a precompiler that inserts into the application source code the statements responsible for gathering and saving its state on a stable storage. The adoption of a checkpoint-based approach introduces, independently of the occurrence of failures, an overhead to the normal application execution time. Integrade checkpointing implementation minimizes this overhead by copying the checkpoint data to a buffer and performing the coding and transfer of checkpoints through a separate application thread, allowing the application to concurrently continue its execution [3].

We developed on Integrade the support for replication, another failure handling technique commonly applied on grid environments. Replication consists on submit the same application with the same set of input parameters a number of times for execution. Each replica represents an active instance of the application, running on a resource different than the other replicas. Thus, as long as not all replicas fail, the application will succeed to execute. When one of the replicas finishes, the grid middleware must discard (or ignore) the others and return the results to the requesting user.

Our implementation also allows the user to customize parameters related to the failure handling mechanism, as part of the application submission process. The user can enable or disable the checkpoint mechanism, set the time interval between consecutive checkpoints, enable or disable the replication mechanism, and set the amount of application replicas to be generated. In this way, flexibility is achieved not only by allowing the customization of different failure handling parameters but also by letting the user combine replication with checkpointing, resulting in four different failure handling techniques: *retrying* (without checkpoint or replication), *checkpointing*, *replication* (without checkpointing), and *replication with checkpointing*.

The resulting Integrade fault tolerance infrastructure is composed by the following components:

- **Execution Manager** (EM): maintains information about each application submission and coordinates the reinitialization of applications in case of failures;

- **Checkpointing library** (ckpLib): provides the functionality to periodically generate checkpoints containing the application state;
- **Autonomous Data Repositories** (ADRs): a distributed stable storage residing on machines that share their resources with the grid;
- **Cluster Data Repository Manager** (CDRM): manages the available ADRs on a cluster and the location of each checkpoint data;
- **Application Replication Manager** (ARM): instantiates the replicas of a single application execution. When the first replica concludes its job, the ARM kills the remaining ones, releasing the allocated grid resources. The Global Resource Manager (GRM) instantiates an ARM on demand for every application submission on which replication is requested.

Integrade checkpointing mechanism adds to the basic application execution protocol (Figure 1) steps responsible for gathering the application checkpoint and its storage in Autonomous Data Repositories (ADRs). A precompiler inserts into the application code the statements responsible for gathering and restoring the application state from a checkpointing library. When the checkpointing library needs to store a checkpoint, it queries the CDRM about available ADRs. Checkpoint data recovery also involves a query to the CDRM, requesting the list of ADRs where the application checkpoints were stored. More details about Integrade checkpointing mechanism can be found in [2], [3].

A. *Integrade Replication Mechanism*

Figure 2 illustrates our implementation of the Integrade protocol for executing application replicas. The protocol rationality minimizes the impact on the code of previously developed components, concentrating the support for replication on a new one: the Application Replication Manager (ARM).

Using the Application Submission and Control Tool (ASCT), the user issues an application execution request informing the amount of replicas that must be generated. The request is forwarded to the GRM (1) that schedules the execution to available resources, notifying the EM that the application has been successfully scheduled (2). The GRM instantiates a new ARM passing the execution request combined with the scheduled grid nodes (3). The ARM obtains the application input files (if any) from the ASCT (4) and forwards the execution request to the LRM running on each scheduled node (5 and 6), generating the application replicas. In this example, only two copies of the application are created. Each LRM downloads the application binary from the Application Repository (7 and 8), requests the application input files from the ARM (9 and 10), and returns to it a notification that the application execution has been accepted (11 and 14). The ARM notifies the ASCT upon the receive of the first acceptance notification from a LRM (12). It also notifies the

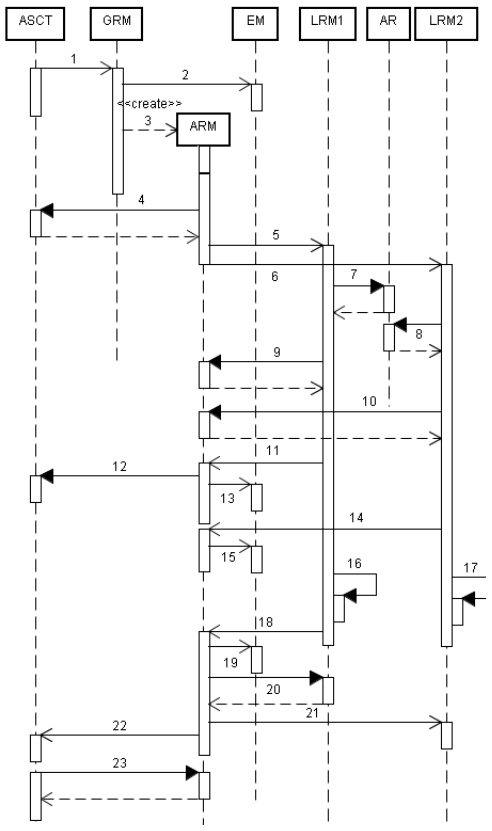

Fig. 2. Execution protocol with replication

EM about each copy of the application execution (13 and 15). Each LRM starts and begin to monitor the application execution on its node (16 and 17), notifying the ARM upon its conclusion (18). The ARM informs the EM about the end of the application execution (19) and downloads the application results (generated output files) (20). If some exception occurs (e.g. the machine is turned off) during the result transfer, the ARM considers as not finished the computation and it waits the terminus of another replica. If no exception occurs, the ARM requests the termination of remaining replicas (21) and notifies the ASCT about the successful application execution (22). Finally, ASCT can download the results from the ARM (23).

V. Performance Evaluation

In order to evaluate the benefits of our flexible fault tolerance mechanism, we performed several experiments that measure the tradeoff among failure handling techniques implemented on Integrade (*retrying* (Rt), *checkpointing* (Ck), *replication* (Rp), and *replication with checkpointing* (RpCk)), considering several different execution environments scenarios. The experiments are similar to the simulations presented by Hwang et al. [8], but were performed in a real Integrade cluster. The cluster was composed by 5 machines with the following configuration: Intel Pentium 4 processor with 2.8 GHz and 1 GB of RAM memory, running Linux 2.6.10, connected by a 100 Mbps Ethernet network. The experiments used the following parameters:

- **Failure-free execution time (F)**: execution time of a task in the absence of failures;
- **Failure Rate (λ)**: random variable representing an arrival rate of failures;
- **Mean Time to Failure (MTTF)**: mean interval between adjacent failures;
- **Downtime (D)**: average time following a failure of a task before it is up again;
- **Number of replicas (N)**: number of replicated tasks, each running on a different machine;
- **Checkpoint Interval (C)**: time interval between two consecutive checkpoints in failure-free runs. If K checkpoints are created during F, then $C = F / K$.

In order to simulate the occurrence of failures, we developed a small application responsible for generating a pseudo random number that represents the moment where the next application failure must occur using an exponential distribution. When the failure moment is reached, a regular application being executed in the grid is abruptly aborted. When replication based techniques were being tested, a uniform distribution was used to select the replica to be aborted.

A first set of experiments used an application called *bootstrap* [4], a generic method to estimate the variability in statistics. The application presented a failure-free execution time (F) of 157.13 seconds.

In the first experiment, we fixed the downtime (D) to 0 seconds and varied the MTTF to 20, 40, 60, 120 and 180 seconds. We used 3 replicas (N) on *replication* based techniques and set the checkpoint interval (C) for *checkpointing* based techniques to 5 seconds.

Table I presents the results obtained. For MTTF equal to 20 seconds and 40 seconds, the techniques *checkpointing* and *replication with checkpointing* presented a better performance than the other two. However, for a MTTF $\geq$ 120 seconds ($\frac{MTTF}{F} \geq 0.76$), *replication* and *replication with checkpointing* were lightly better than *checkpointing* and considerable better than *retrying*.

TABLE I
BOOTSTRAP APPLICATION COMPLETION TIME (IN SECONDS) FOR EACH FAILURE HANDLING TECHNIQUE (DOWNTIME = 0 S)

MTTF	*Rt*	*Ck*	*Rp*	*RpCk*
20	∞	184.55	324.78	162.16
40	∞	172.30	175.81	158.72
60	899.24	167.07	160.26	157.81
120	408.89	161.36	157.14	157.76
180	229.62	159.87	157.13	157.51

In a second experiment, we used the same values for N (3) and C (5 seconds), altering the downtime value to D = 79 seconds ($\approx \frac{F}{2}$). We also varied the MTTF to 20, 40, 60, 120 and 180 seconds. As Table II shows, when the downtime increases to $\approx \frac{F}{2}$, *replication* and *replication with checkpointing* outperformed the other two techniques.

TABLE II
BOOTSTRAP APPLICATION COMPLETION TIME (IN SECONDS) FOR EACH FAILURE HANDLING TECHNIQUE (DOWNTIME = 79S $\approx \frac{F}{2}$)

MTTF	*Rt*	*Ck*	*Rp*	*RpCk*
20	∞	899.06	527.24	280.82
40	∞	450.85	241.01	181.67
60	1117.59	408.61	180.12	158.22
120	626.02	286.75	157.11	157.70
180	319.55	246.81	157.06	157.61

In a second set of experiments, we adopted a matrix multiplication application. Each matrix consists of 1700×1700 elements of type float. The application presented a failure-free execution time (F) of 115.75 seconds. The experiment objective is to evaluate the impact of the checkpoint data size, since the matrix multiplication application generates a huge checkpoint of 67.8 MB, in contrast to the *bootstrap* application, whose checkpoint data has only 274.82 KB.

In a first experiment, we fixed the downtime (D) to 0 seconds and varied the MTTF to 15, 20, 30, 60 and 100 seconds. N was set to 3 and C to 5 seconds. Table III shows that in an environment with high failure rate (MTTF equals 15 and 20 seconds), *checkpointing* and *checkpointing with replication* presented a better performance, while for MTTF $\geq$ 30 seconds ($\frac{MTTF}{F} \geq 0.26$), the use of those techniques is inappropriate, due to the overhead caused by the checkpointing capture.

TABLE III
MATRIX MULTIPLICATION APPLICATION COMPLETION TIME (IN SECONDS) FOR EACH FAILURE HANDLING TECHNIQUE (DOWNTIME = 0 S)

MTTF	*Rt*	*Ck*	*Rp*	*RpCk*
15	∞	181.28	255.44	154.52
20	∞	173.38	184.13	154.20
30	877.29	166.80	135.18	152.33
60	429.60	157.64	117.37	150.16
100	228.10	155.59	116.00	149.74

Altering the downtime value to D = 57 seconds ($\approx \frac{F}{2}$), we obtained the results presented on Table IV. As we can see, when the downtime increases to $\approx \frac{F}{2}$, *replication* and *replication with checkpointing* outperformed the other two techniques.

TABLE IV
MATRIX MULTIPLICATION APPLICATION COMPLETION TIME (IN SECONDS) FOR EACH FAILURE HANDLING TECHNIQUE (DOWNTIME = 57S $\approx \frac{F}{2}$)

MTTF	*Rt*	*Ck*	*Rp*	*RpCk*
15	∞	788.97	505.37	261.87
20	∞	724.90	197.87	192.76
30	∞	430.67	164.95	188.12
60	772.51	346.73	125.02	168.03
100	407.83	287.38	116.33	157.24

Considering the experimental results and the objective of minimizing the application response time, we can conclude that in environments with high fault rates (low MTTF), the use of checkpointing presented the best results, specially when combined with replication. If the execution environment presents a low fault rate (high MTTF), the use of replication becomes more attractive. In environments with low fault rate and low downtime, the use of checkpointing can be consider a good choice for applications with small checkpoint size, since the response time is very close to the one obtained with replication, with the advantage of using less grid resources. As the application checkpoint size or the environment downtime increase, replication becomes more attractive.

VI. Related Work.

The Condor project provides fault tolerance for grid applications through a checkpoint approach [11]. Checkpointing at Condor is "transparent": developers do not need to write specific checkpointing code. Checkpoints are stored in files on the local executing machine or in a checkpoint server.

OurGrid [1] uses task replication to provide fault tolerance and for improving performance. OurGrid does not support checkpoint natively, but users can optionally use third part software (e.g. a checkpoint library). OurGrid also provides the necessary infrastructure to manage the checkpoint usage, such as a stable storage. The checkpointing approach, although, is not transparent.

CoordAgent [5] is a mobile-agent-based middleware. It provides a checkpointing mechanism that inserts state-capturing functions into a user source code through a language preprocessor. CoordAgent uses two compiler-compiler tools: ANTLR for C/C++ and JavaCC for java source code, since both languages are supported for application development.

There are several differences between Integrade fault tolerance mechanisms and the ones described above, such as the use of a distributed stable storage and provision of checkpointing for parallel BSP applications (not presented on this article for shortness reasons). We would like to highlight the flexibility of Integrade approach that allows the user to combine different fault-tolerance techniques.

Hwang et al. [8] present a flexible fault tolerance framework for grids environments. The framework uses a workflow approach, where users can specify failure handling parameters using a XML based language called WPDL (*workflow Process Definition Language*). The failure handling framework, Grid-WFS, provides support for multiple failure recovery techniques (retrying, checkpointing, replication and replication with checkpointing). Grid-WFS checkpointing mechanism is not transparent, in contrast with Integrade. Integrade also provides a distributed stable storage.

VII. Conclusion and Ongoing Work

This paper presented a flexible fault-tolerance mechanism implemented on Integrade grid middleware. Flexibility is achieved not only by allowing the customization of different failure handling parameters but also by letting the user combine replication with checkpointing, resulting in four different failure handling techniques: *retrying* (without checkpoint or replication), *checkpointing*, *replication* (without checkpointing), and *replication with checkpointing*.

We evaluated the benefits of our flexible mechanism by performing several experiments that measure the tradeoff among the failure handling techniques implemented on Integrade. Considering the objective of minimizing the application response time, the results demonstrated that the best failure handling technique varied as we altered environment parameters such as the MTTF and the downtime, sustaining the conclusion that grid middlewares can benefit from providing different failure handling strategies.

Towards our objective of developing an autonomic grid middleware, we are currently implementing the support for automatic decision of the failure handling technique to be applied in case of failure, given an application submission. Integrade will choose the best failure handling technique considering the current grid execution environment. We are augmenting Integrade with the Adapta framework, a reflective middleware that provides support for developing self-adaptive component-based distributed applications.

References

[1] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An approach to easily assemble grids with equitable resource sharing. In *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2003.

[2] R. Y. de Camargo, R. Cerqueira, and F. Kon. Strategies for Checkpoint Storage on Opportunistic Grids. *IEEE Distributed Systems Online*, 7(9), September 2006.

[3] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. In *ACM/IFIP/USENIX 2nd International Workshop on Middleware for Grid Computing*, Toronto, Canada, October 2004.

[4] B. Efron and R. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, New York, 1993.

[5] M. Fukuda, Y. Tanaka, L. F. Bic, and S. Kobayashi. A mobile-agent-based pc grid. *IEEE Computer*, 2003.

[6] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. Integrade: Object-oriented grid middleware leveraging idle computing power of desktop machines. *Concurrency and Computation: Practice & Experience. Vol. 16, pp. 449-459*, 2004.

[7] S. Hariri, B. Khargharia, H. Chen, J. Yang, Y. Zhang, M. Parashar, and H. Liu. The autonomic computing paradigm. *Cluster Computing: The Journal of Networks, Software Tools, and Applications, Kluwer Academic Publishers*, 8(5), 2005.

[8] S. Hwang and C. Kesselman. Gridworkflow: A flexible failure handling framework for the grid. *hpdc*, 00:126, 2003.

[9] J. Kaufman, T. Lehman, G. Deen, and J. Thomas. Optimalgrid: autonomic computing on the grid. http://www-128.ibm.com/developerworks/library/gr-opgrid/.

[10] P. Leong, C. Miao, and F. Sung. Agent mediated autonomic service orchestration in grid environment. In *3rd IEEE International Conference on Industrial Informatics*, August 2005.

[11] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical report, University of Wisconsin-Madison, April 1997.

[12] P. Townend and J. Xu. Fault Tolerance within a Grid Environment. *In Proceedings of AHM2003*, page 272, 2003.