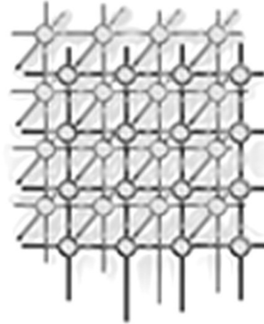
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



# Adaptive fault tolerance mechanisms for opportunistic environments: a mobile agent approach

V. G. Pinheiro†, A. Goldman, F. Kon
{vinicius, gold, kon}@ime.usp.br

*Department of Computer Science, Rua do Matão, 1010, Cidade Universitária, 05508-090, São Paulo - SP, Brazil*

**SUMMARY**

**The mobile agent paradigm has emerged as a promising alternative to overcome the construction challenges of opportunistic grid environments. This model can be used to implement mechanisms that enable application execution progress even in the presence of failures such as the mechanisms provided by the MAG middleware (Mobile Agents for Grids). MAG includes retrying, replication and checkpointing as fault tolerance techniques; they operate independently from each other and they are not capable of detecting changes on resource availability. In this paper, we describe a MAG extension that is capable of migrating agents when nodes fail, which optimizes application progress by keeping only the most advanced checkpoint, and also migrates slow replicas. The proposed approach was evaluated via simulations and experiments, which showed significant improvements.**

KEY WORDS: opportunistic grid; mobile agent; adaptive fault tolerance

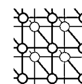
## 1. INTRODUCTION

Opportunistic grids are distributed environments built to leverage the computational power of idle resources geographically spread across different administrative domains. These environments comprise many characteristics such as high level of heterogeneity and large changes on resource availability.

In distributed systems, failures can occur due to several factors, most of them related to resource heterogeneity and distribution. These failures together with the use of the resources

†Vinicius Pinheiro is partially supported by a graduate fellowship from CAPES, Brazil

Copyright © 2000 John Wiley & Sons, Ltd.
*Received 31 January 2000*
*Revised 19 September 2002*



by its owners modify grid resource availability (i.e., resources can be active, busy, offline, crashed, etc.). The opportunistic grid middleware should be able to monitor and detect such changes, rescheduling applications across the available resources and dynamically tuning the fault tolerance mechanisms to better adapt to the execution environment.

In this work, we implemented dynamic fault tolerance mechanisms based on task replication and checkpoints for grid applications. A task replica is a copy of the application binary that runs independently from the other copies. Through these mechanisms, the middleware is capable of migrating tasks when nodes fail. It coordinates task replicas and its checkpoints in a rational manner, keeping only the most advanced checkpoint and migrating slow replicas. These features dynamically improves application execution, compensates the misspend of resources introduced by the task replication and solves scalability issues.

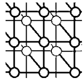
These mechanisms compose a feedback control system [1, 2], gathering and analyzing information about the execution progress and adjusting its behavior accordingly. To build these mechanisms, we rely on the mobile agent paradigm [3]. Mobile agents are programs that can move from one resource to another in an autonomous way, carrying its data and execution state and resuming its execution at the destination. We argue that agents are suitable for opportunistic environments due to intrinsic agent characteristics such as:

1. *Cooperation*: agents have the ability to interact and cooperate with other agents; this can be explored for the development of complex communication mechanisms among distributed application tasks;
2. *Autonomy*: agents are autonomous entities, meaning that their execution goes on without any or with little intervention from the clients that started them. This is a suitable model for submission and execution of grid applications;
3. *Heterogeneity*: most mobile agent platforms can be executed in heterogeneous environments, an important characteristic for better use of computational resources across multi-organization environments;
4. *Reactivity*: agents can react to external events such as variation on resources availability;
5. *Mobility*: agents can migrate from one node to another, moving part of the computation being executed, helping to balance the load on grid nodes.

Since 2004, our research group has been using the agent paradigm to develop a grid software infrastructure, leading to the MobiGrid [5] and MAG [6] projects. These projects are based on the InteGrade middleware [7, 23], which follows an opportunistic approach, where workstations idle computing power is used for executing computationally-intensive parallel applications.

This work describes enhancements to the MAG middleware that address the dynamism of opportunistic grids, managing fault-tolerant execution and resource allocation for sequential and embarrassingly parallel applications. In the next section, we present the related work. In Section 3, we present the MAG architecture and its fault tolerance mechanisms. In Section 4, we describe the implementation of the dynamic replication and unified checkpointing mechanisms. We describe simulation results and experiments that assess our proposal in sections 5 and 6. Finally, in the last section, we present our conclusions and future work.

Copyright © 2000 John Wiley & Sons, Ltd.





## 2. RELATED WORK

The most well-known work in the field of opportunistic grid is provided by the SETI@home project [29] whose research is focused on finding signs of extraterrestrial life by processing signals received by radio telescopes. This type of application is embarrassingly parallel (also known as bag-of-tasks or parametric applications) since the input data can be divided into smaller parts that are distributed and processed by workstations volunteers.

The success of the SETI@home project and the emergence of similar projects as GIMPS [32], Climateprediction.net [34] and Einstein@home [33] motivated the development of BOINC [31], a software platform for volunteer computing that allows a single workstation to participate in various projects.

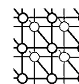
Anybody can download the BOINC client and integrate their machine to the grid. Therefore, more attention is paid on security aspects and on the reliability of the results. The detection of incorrect or corrupted data is conducted through the submission of multiple copies of the same work unit. The state of the tasks is stored periodically at the user's machine. Our approach also uses task replication and checkpointing, however, for different purposes. Task replication is used both as a strategy for tolerating failures as well as to accelerate the execution. Moreover, the checkpointing is performed remotely, allowing the comparison of the replicas execution progress, which is crucial for our approach, as we shall see later.

Another bag-of-tasks approach is based on OurGrid [10], a computational grid that allows laboratories to share the idle cycles of its resources through a network of favors, promoting the fair division of processing time between the entities of this grid. This grid provides task replication through the Workqueue with Replication scheduler (WQR) [11] in a similar way to our work. However, it lacks checkpointing support. The user must use an external library since the project does not provide an automatic way to instrument the application binary.

Several works deal with checkpointing techniques to guarantee the progress of sequential long running applications. One that is directly related to our work is the Grid-WFS framework [12], where the authors studied several approaches to deal with failures on machines. The handling techniques were: retrying, checkpointing, replication, and replication with checkpointing. They concluded that in grid environments with high downtime, as in some opportunistic environments, the replication with checkpointing outperforms the other ones, using as comparison the lower completion time. The Condor project also provides some fault tolerance mechanisms to deal with unstable and opportunistic environments: checkpointing and process migration [13]. However, Condor does not perform task replication, which could be used to improve application execution progress in the presence of host and network failures.

In the context of mobile agents, some works stand out. A few of them uses opportunistic contexts (e.g. UWAgents [20]), but most of them presents characteristics more related to the middleware, not the application (e.g. ARMS [15] and the works published by Loke [22] and Martino and Rana [21]). Some of the mobile agents work were done within our project InteGrade [7]. The first ideas of using mobile agents on an opportunistic grid appeared in [5] where an architecture based on Aglets [30] is first presented, and then evaluated with the use of several replicas in [4]. More recently, a work based on the mobile agents framework JADE [26] was also presented [6], where there is application instrumentation, to provide transparent checkpointing and some work on fault tolerance.

Copyright © 2000 John Wiley & Sons, Ltd.



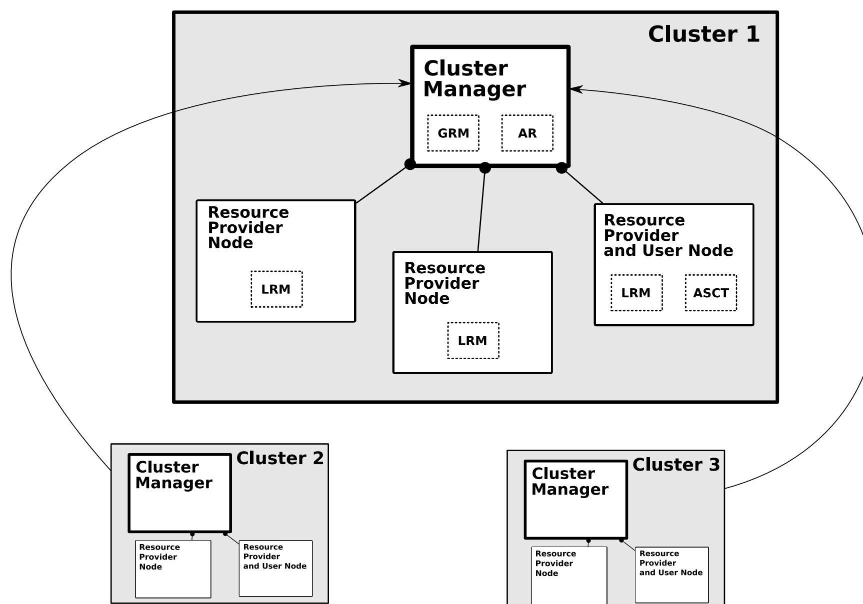



Figure 1. InteGrade architecture

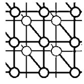
To the best of our knowledge, this paper is the first that specifically uses a mobile agent approach to bind task replication and checkpointing within a grid middleware, providing dynamic fault tolerance mechanisms for sequential and parametric applications on opportunistic environments.

## 3. THE INTEGRADE/MAG MIDDLEWARE

The InteGrade project [7, 23] involves the development of a grid middleware that leverages the idle computational power of desktop machines. Its architecture follows a hierarchy in which each node can assume different responsibilities. The *Cluster Manager* is represented by one or more nodes that are responsible for managing that cluster and performing communication with other clusters. A *Resource Provider* node exports part of its resources, making them available to grid users. A *User Node* belongs to a grid user who submits grid applications. As we can see in Figure 1, the InteGrade architecture follows a two-tier intra-cluster hierarchy combined with an inter-cluster network.

The MAG project [6] introduces the mobile agent technology as a new way of executing applications on InteGrade. Through MAG, the grid user can submit Java applications, not previously supported by the native InteGrade middleware. This is performed by dynamically

Copyright © 2000 John Wiley & Sons, Ltd.





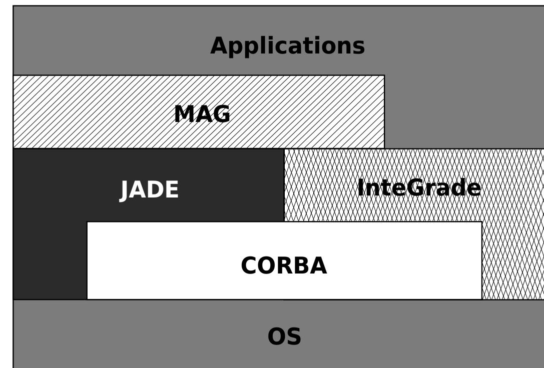



Figure 2. Layered view of InteGrade/MAG middleware

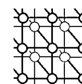
loading sequential grid applications into mobile agents. MAG uses JADE (*Java Agent Development Framework*) [26] as the agent platform to provide agent services such as communication and life cycle monitoring. In JADE, each agent has a private message queue and the agent communication is performed through message exchanges written in the Agent Communication Language (ACL), compliant with the FIPA (Foundation for Intelligent Physical Agents) standard specification [14]. This feature avoids race conditions since the messages can be read in an asynchronous way, and are processed one at a time*.

In Figure 2, we have a layered view of MAG's infrastructure. The InteGrade middleware is used as an implementation base for MAG. The JADE layer provides communication features, life cycle management and monitoring of mobile agents. The CORBA layer provides naming service for JADE and InteGrade components. The InteGrade/MAG is multiplatform as MAG is implemented in Java and InteGrade follows the IEEE POSIX [25] specifications. Thus, the OS layer can be performed by several operating systems.

To avoid duplication of efforts, the MAG project was built on top of InteGrade components, namely, the Global Resource Manager (GRM), the Local Resource Manager (LRM), the Application Repository (AR) and the Application Submission and Control Tool (ASCT) (see Figure 1). The GRM is the main grid component and it is executed in the Cluster Manager Nodes; it holds information about the registered LRMs and it is able to dispatch tasks to them. The LRM is executed in each Resource Provider node; it loads the execution environment and executes tasks submitted to them. The AR provides a cluster repository to store application binaries. Finally, the ASCT provides a user interface for grid application submission, monitoring and collection of computation results.

*The agents in the JADE platform are single-threaded

Copyright © 2000 John Wiley & Sons, Ltd.



In addition, the MAG architecture adds components that provide mobile agents capabilities and fault tolerance mechanisms:

1. The *ExecutionManagementAgent (EMA)* stores information about current and past executions such as current execution state (accepted, running or finished), input arguments and scheduled machines. This information can be retrieved to restore applications to the point they were before the failure;
2. The *AgentHandler* runs on top of the LRMs and works as a proxy to the JADE agent platform, instantiating MAGAgents for each requested execution;
3. The *ClusterReplicationManagerAgent (CRM)* receives requests for execution with replicas from the GRM and creates an ERM agent to handle the request;
4. The *ExecutionReplicationManagerAgent (ERM)* distributes the replicas across the LRMs in the distributed system;
5. The *StableStorage* agent receives the compressed checkpoints, storing them in the file system and retrieving them when prompted. This agent runs in the Cluster Manager node;
6. The *MAGAgent* is the MAG main component; it wraps the application, instantiates it, and catches its exceptions. It also controls the application life cycle;
7. The *AgentRecover* is created on demand by the MAGAgents to recover an execution state in the presence of failures.

### 3.1. Fault-Tolerance in MAG

The MAG fault tolerance mechanisms can be combined to meet different scenarios of resource availability, resulting in four different strategies:

1. *Retrying*: every time the application fails (by throwing an runtime exception), its agent migrates to another node and restarts the execution;
2. *Replication*: multiple application replicas are submitted for execution at the same time. When one of the replicas finishes, its agent sends a message to the CRM to discard the other replicas. In case of failure, retrying is applied;
3. *Checkpointing*: the MAGAgent periodically saves the execution state of its application by sending a message to the StableStorage agent. In case of application failure, retrying is applied, but the execution is resumed from a checkpoint;
4. *Checkpointing with Replication*: the execution state of each replica is periodically saved in the StableStorage agent. Retrying and resuming of execution are applied independently for each replica in case of failures.

Currently, the MAG middleware supports only the submission of embarrassingly parallel and sequential applications. This is implemented by extending the `MagApplication` class; the middleware then wraps the application code into a mobile agent and submits it to the agent platform. The checkpoint mechanism implementation is based on Java code instrumentation provided by the *MAG/Brakes* framework [19].

The following figure is a description of what happens when a submission with task replication is required in MAG (see Figure 3):

Copyright © 2000 John Wiley & Sons, Ltd.

*Prepared using* cpeauth.cls

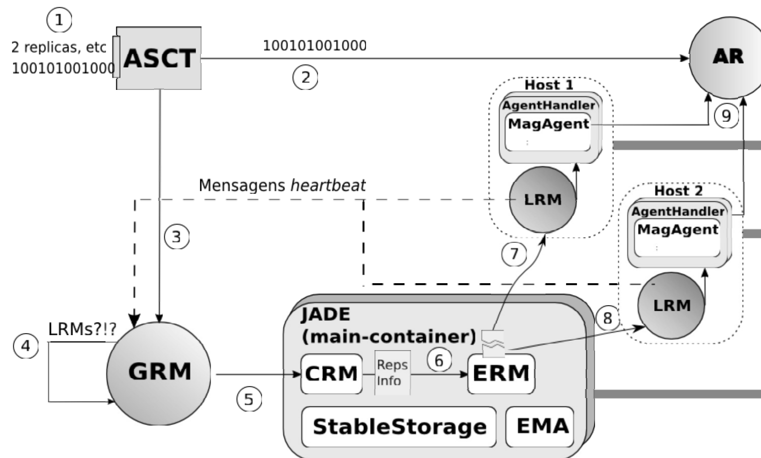



Figure 3. Application submission on MAG

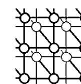
The user submits an application through the ASCT interface, along with additional information (1) such as input data, number of replicas, input and output files, etc. The user can also specify if one machine can process more than one task at the same time. The binary is stored in the AR (2). The execution request is sent to the GRM (3).

After submission, the GRM checks if there are enough resources according to execution constraints provided by the user (e.g. if number of replicas is allowed or not to exceed the number of available LRMs) (4). If so, the GRM delegates execution to the CRM (5). This component processes specific information to each replica to be generated (e.g. replicates the input arguments and assigns identifiers to each replica) and creates an ERM agent to manage the request (6). The ERM proceeds the execution by passing to each LRM the execution information related to one of the replicas (7). Thereafter, each LRM delegates the execution to the AgentHandler, which creates a MAGAgent to encapsulate the task (8). The MAGAgent downloads the binary from the AR (9), instantiates the application, and notifies the AgentHandler when execution is completed.

## 4. IMPROVING MAG: TOWARDS AN ADAPTIVE MIDDLEWARE

As shown in Section 3.1, the MAG middleware supports multiple fault tolerance techniques, but these techniques operate solely. Besides, they do not perform any automatic adjustments to adapt themselves to changes in resource availability. Events such as network partitioning, crash failures, machine shutdowns, nodes joining the grid and nodes leaving the grid define the resource availability of the executing environment. Thus, it is desirable that the middleware

Copyright © 2000 John Wiley & Sons, Ltd.



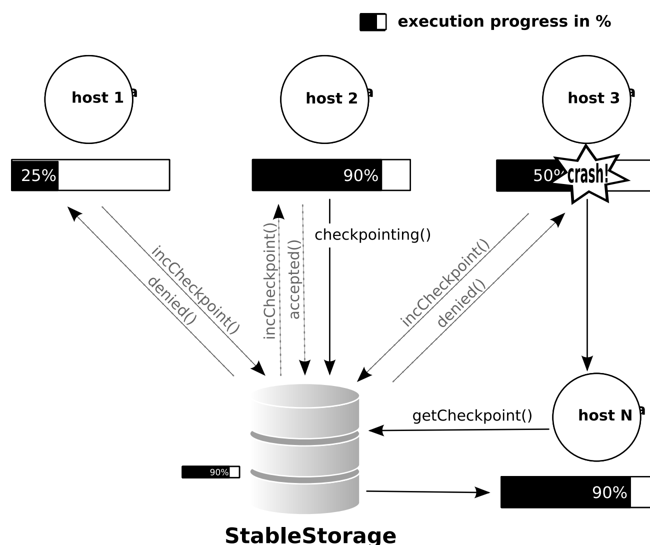



Figure 4. Unified Checkpoint model

includes fault tolerance mechanisms to adapt dynamically to these changes, providing a better quality of service for grid users.

### 4.1. Unified Checkpoint

Since the MAG fault tolerance mechanisms work independently from each other, this model does not scale well because it makes all replicas perform checkpointing periodically. This increases the communication traffic between the resource provider nodes and the StableStorage agent, consuming more grid resources. Another disadvantage of this model is related to resource heterogeneity: in a heterogeneous environment like opportunistic grids, some replicas will advance its execution faster than others. If the most advanced replica crashes in a way that MAG cannot detect, its latest checkpoint will not be used by the slower replicas and part of the execution will be lost.

To solve this problem, we propose a mechanism named Unified Checkpoint. In this new model, the replicas periodically send information about their execution progress and only the most advanced replica is authorized to perform checkpointing. To enable this feature, the applications must invoke the method `incCheckpoint()` that increases a progress counter.

Copyright © 2000 John Wiley & Sons, Ltd.

It is up to the application programmer to choose the most appropriate places to put these invocations into the source code since this is a very application-specific issue[†].

When the replica hits a checkpoint, it sends only the progress counter and the StableStorage compares this value to the ones sent by the other replicas. Only the replica with the highest counter value is queried to perform the checkpoint. This model is depicted in Figure 4.

In this figure, the replica running on host 2 is the most advanced one. When the replica running on host 3 crashes, the MAG recovery mechanism is executed: a new replica is created on host N and the StableStorage is queried for the checkpoint. The checkpoint stored by the most advanced replica is the only option and so it is sent to the new replica, which resumes its execution from this advanced stage.

### 4.2. Replica Replacement

Although the checkpointing and the replication of tasks now operate together to form a more integrated fault tolerance system, some events such as machine crashes, may reduce the number of active replicas. In addition, it would be interesting to compare the replica progress counters to detect slow replicas and decide whether they should be moved to another, hopefully faster, computing node.

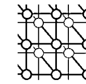
To accomplish that, we propose a feedback control system based on periodical analysis of resource availability. This system is depicted in Figure 5.

Initially, the grid user sends an application to the grid. The application replicas are created and submitted to execution on the grid nodes. The number of replicas created is equal to a fixed number defined by the user, but respecting a maximum value of replicas for each application, a value that can be defined by grid administrators. While running, these replicas are susceptible to failures related to intrinsic characteristics of opportunistic environments such as network partitions, machine shutdowns, out-of-memory errors, etc. These failures reduce the number of executing replicas and also modify the amount of available resources. These changes are detected by the system that, after a period without getting responses from the crashed/offline nodes, updates the list of nodes that are still alive (and the new ones that have joined or rejoined the grid recently). Furthermore, during this time, some replicas in machines with limited resources may become delayed. Based on this information, new replicas are created and the ones that are running slowly are migrated to new nodes. The Unified Checkpoint is present throughout this process: new replicas resume their execution starting from the checkpoint of the most advanced replica. This mechanism works even when the most advanced replica crashes, as its last checkpoint remains stored at the StableStorage so that the new replica can resume from it.

The StableStorage is a central component of our system. If it crashes, the checkpoints become unavailable. To avoid this to happen, we can adopt two solutions already implemented, but yet

[†]Misplacing the `incCheckpoint()` invocation could degrade the performance of the Unified Checkpoint. However, we assume that it should be easy for the developer of the application to detect a good place to put the call. Otherwise, normal checkpointing must be used rather than Unified Checkpoint if one is not sure about where to put the invocations and do not want to take that risk.

Copyright © 2000 John Wiley & Sons, Ltd.



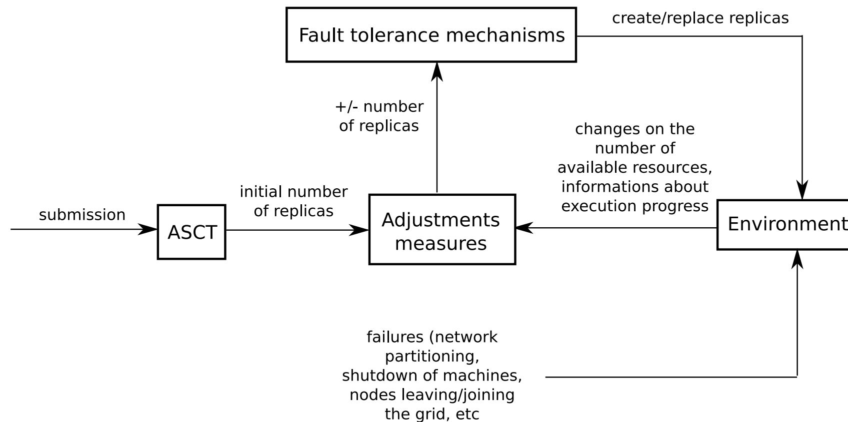



Figure 5. Dynamic replication: a feedback system model

to be integrated. One solution is provided by the JADE platform. JADE supports synchronous replication of containers [27]. Through this service, it is possible to instantiate several copies of the StableStorage in a way that, when the main copy receives a message, this message is broadcast to the remaining instances to keep them synchronized. If the main copy crashes, another instance takes its place.

Another solution is provided by the OppStore middleware [8, 9]. The OppStore is an InteGrade component for distributed storage that uses the free space available on the machine disks of a grid. These machines must be organized in clusters connected by a peer network as in the InteGrade architecture. Before being stored, the data is coded in redundant fragments in a way that the data can be restored from a subset of these fragments.

## 5. SIMULATIONS

In this section, we describe a series of event-based simulations in various scenarios, demonstrating the potential value of adding dynamic fault tolerance mechanisms to MAG. Our analysis focuses on task execution times and resources consumption. To run our simulation scenarios, we used the GridSim toolkit [17]. The parameters used in our simulation (mostly borrowed from previous works by Plank and Elwasif and by Beguelin et al. [18, 16]) follow.

- *Failure rate ($\lambda$)* is a random variable representing an arrival rate of failures governed by a Poisson distribution. TBF (time between failures) is a random variable governed by an exponential distribution with MTBF representing the mean;

Copyright © 2000 John Wiley & Sons, Ltd.

*Prepared using* cpeauth.cls

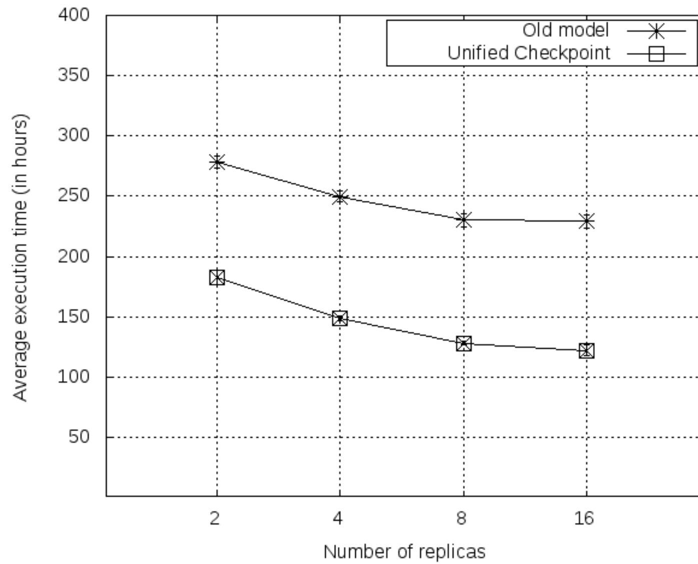



Figure 6. Performance comparison: Unified Checkpoint versus old model

- *Downtime (D)* is the average time following a failure of a task before it is up again, governed by an exponential distribution;
- *Number of replicas (N)* is the number of copies of an application, with each one running on a different machine;
- *Delay ratio ($\gamma$)* represents the ratio between the progress counters of the most advanced replica and the other replicas. This ratio is used to replace delayed replicas.

We simulated a cluster environment with 100 heterogeneous machines connected by a 100Mbps network. The processing power of the resources were generated randomly with a uniform distribution from 800 to 1600 MIPS, which are typical values for the SPECfp benchmark [28] for opportunistic grid machines at the time of this writing.

We used three parameters to model the tasks: number of instructions in MI (millions of instructions), binary size (in bytes) and output file size (in bytes). In our experiments, we chose to simulate long running and short running tasks and, to do so, we set the long task length to $6.048 \times 10^8$ MI; the short ones have length ten times smaller than the long ones: $6.048 \times 10^7$ MI. The binary size is 320 Kilobytes and the output file size is 15.6 Kilobytes, numbers that were taken from a sample application we run on InteGrade. If we consider the most powerful machine allowed in our experiments, it would take 105 hours to execute a long task and 10.5 hours to execute a short task.

Copyright © 2000 John Wiley & Sons, Ltd.

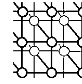


Table I. Average number of substitutions for long running applications

| Replicas | Num. of subst. ($\gamma$: 0.5) | Num. of subst. ($\gamma$: 0.9) |
|---|---|---|
| 2 | 2.0 $\pm$0.16 | 10.5 $\pm$0.60 |
| 4 | 4.1 $\pm$1.15 | 28.7 $\pm$5.55 |
| 8 | 10.3 $\pm$3.51 | 68.1 $\pm$19.22 |
| 16 | 30.2 $\pm$4.40 | 124.0 $\pm$31.42 |

We measured the task execution times to compare the performance of the proposed techniques against the old model. We used 2, 4, 8 and 16 replicas, and fixed 60 minutes as the MTBF value to obtain a $\lambda$ of 24 failures per day[‡]. Downtime (the $D$ parameter) was fixed to 30 minutes. These values were used to simulate a typical opportunistic environment for distributed processing such as student laboratories, where machines are regularly turned off and rebooted. For each number of replicas, we performed 40 simulations, measuring task execution times, computing the arithmetic mean, and the 95% confidence interval with a t-Student distribution. We assumed that the application execution time is the execution time of the replica that finishes first. The results are plotted in Figure 6, Figure 7 and 8.

First of all, it becomes clear that increasing the number of replicas results in shorter execution times in both strategies. But we can see a considerable gain in the total execution time when using the dynamic strategy presented in this paper. The potential advantage of adopting the Unified Checkpoint mechanism occurs independently of the number of replicas used in our simulation. In all cases, the Unified Checkpoint outperforms the old model obtaining better execution times (at least 34% faster). This difference increases as the number of replicas increases, achieving its maximum performance improvement when 16 replicas were submitted (execution time 47% lower). In the simulated scenarios, which are common in the field of High-Performance Computing, the amount of time saved when using the Unified Checkpoint varied between 95 and 107 hours, which means getting the computation results about 4 days earlier.

In these simulations, the replacement of delayed replicas occurred with $\gamma = 1/2$ since we established that no replica should be more than 50% behind of the unified checkpoint. However, for a better understanding of the replica replacement mechanism, we also made simulations with $\gamma = 9/10$ to increase the number of substitutions. Figure 7 shows the results for both scenarios of replica replacement in the Unified Checkpoint model. Those simulations were carried out in absence of failures since our goal was to observe only the replica replacement mechanism. Failures would activate the replica recovery mechanism and we wanted to avoid that. Table I shows the average number of substitutions for each $\gamma$ used (along with its respective standard deviations).

[‡]This means that during a day, in average 24 failures will occur considering all machines of the environment.

Copyright © 2000 John Wiley & Sons, Ltd.

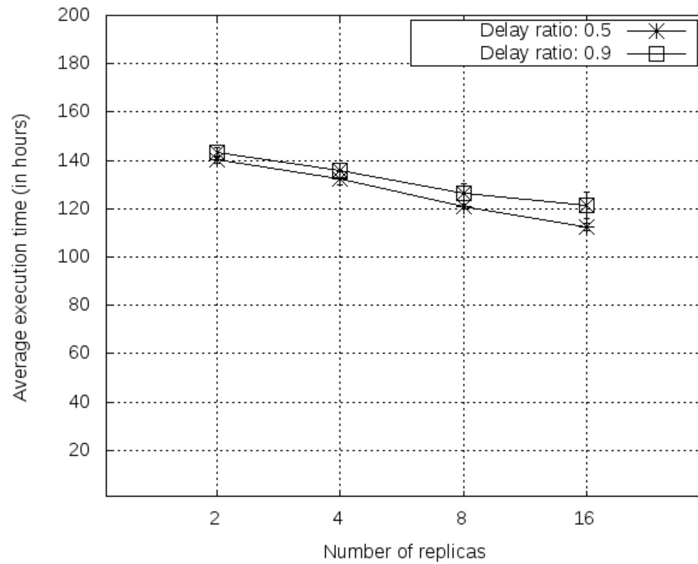


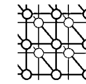

Figure 7. Execution time for different delay ratios

As we can see, there is no significant difference in the execution time, although the average number of substitutions with $\gamma = 0.9$ was higher for all numbers of replicas used. Changing the $\gamma$ to a value closer to 1 increased the number of substitutions substantially (between 400% and 700% of increase in the observed cases). But the results suggest that, from a certain point, increasing the number of replica substitutions does not lead to smaller execution times.

We also compared the old and new models with few machines to see what happens in a more competitive scenario when a machine must run more than one application replica at the same time. These simulations considered no failures since our goal was only to observe how the resource competition would affect execution time. In these simulations, we used short tasks and the results are depicted in Figure 8.

It is interesting to note that no large changes happened while the number of replicas remained between 2 and 16. This happened because not all machines needed to process more than one replica and the replicas that had one machine just for them brought down the average execution time. The large changes occurred only when the number of replicas exceeded the number of machines. At that point, from 16 to 32 replicas, the execution time increases more than 200% because each replica had to compete with other replicas for the same machine. But the advantage of the Unified Checkpoint model remains, independently of the number of used replicas.

Copyright © 2000 John Wiley & Sons, Ltd.



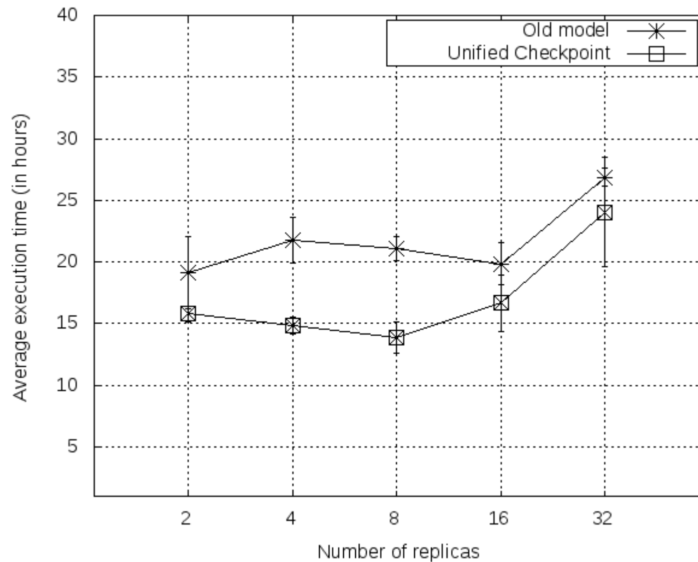



Figure 8. Unified Checkpoint versus old model with 10 machines and short tasks

## 6. EXPERIMENTS

In this section, we present the results of some experiments performed in a small, controlled and real environment. We evaluated the impact on memory consumption and processing performance caused by the introduction of the Unified Checkpoint.

The experiments were performed in an environment composed of heterogeneous and non-dedicated desktop machines located in two laboratories of the Institute of Mathematics and Statistics of the University of São Paulo. In total, 17 machines were used: six are located in the LCPD (Laboratory for Parallel and Distributed Computing) and 11 are located in the Eclipse Laboratory. These machines run Linux operating system. They are connected by a Fast Ethernet 100Mbps place and have their settings displayed in Table II.

To simulate a machine local workload, as it would be found in an opportunistic environment, we scheduled the execution of a script in each machine. The script is a loop with two variations that differ by the amount of processing power consumed. One variation is two times heavier than the other. Of course, the fraction of the processing power consumed by the script varies according to each machine specification, but in all machines used on the experiment, this fraction was above 25%. This number corresponds to the average fraction observed when the students were using these machines for ordinary activities like web browsing and word processing. Following the same reasoning, we scheduled the script to be executed in peak time – from 8 a.m. to 12 a.m. and from 2 p.m. to 8 p.m. – and, for each time the loop was executed,

Copyright © 2000 John Wiley & Sons, Ltd.

*Prepared using* cpeauth.cls

Table II. LCPD and Eclipse machines

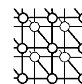
| LCPD | | | | | |
|---|---|---|---|---|---|
| Machine | Processor | RAM/Swap | OS/Arch | Kernel Version | Distribution |
| villa | AMD 2.0 GHz | 1 GB/1.5 GB | Linux i686 | 2.6.22-14-generic | Ubuntu 7.10 (gutsy) |
| ilhabela | AMD 2.0 GHz | 1 GB/1.5 GB | Linux i686 | 2.6.22.14-generic | Ubuntu 7.10 (gutsy) |
| taubate | AMD 2.0 GHz | 3 GB/768 MB | Linux x86_64 | 2.6.22.14-generic | Ubuntu 7.10 (gusty) |
| giga | Intel 3.0 GHz | 2 GB/2 GB | Linux i686 | 2.6.22.14-generic | Debian 5.0 (lenny) |
| orlandia | AMD 2.0 GHz | 1 GB/640 MB | Linux i686 | 2.6.22.14-generic | Ubuntu 7.10 (gutsy) |
| motuca | AMD 2.2 GHz | 1.5 GB/2 GB | Linux x86_64 | 2.6.10 | Debian 5.0 (lenny) |
| Eclipse | | | | | |
| mercurio | AMD 1.4 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| venus | AMD 1.4 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| terra | AMD 1.4 GHz | 1 GB/1.5 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| marte | AMD 2.0 GHz | 1 GB/2 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| jupiter | AMD 1.4 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| saturno | AMD 1.4 GHz | 1 GB/1.2 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| urano | AMD 1.4 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| netuno | AMD 1.4 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| plutao | AMD 1.4 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| hubble | AMD 1.4 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-9-generic | Ubuntu 8.10 (intrepid) |
| callisto | AMD 1.5 GHz | 1 GB/0 GB | Linux i686 | 2.6.27-7-generic | Ubuntu 8.10 (intrepid) |

the variation was chosen at random from a uniform distribution. We set the host *orlandia* to be the cluster manager and the other machines to be the resource providers.

In our experiments, we used a Java application that calculates the approximate value of $\pi$ iteratively using a statistical approach. This application makes extensive use of processing power and makes several calls to the same method every second. We placed the checkpoint invocation after this method. Thus, as seen in section 3.1 about the checkpointing mechanism and considering an interval of approximately 5 seconds between the checkpoints, this application makes extensive use of the checkpointing mechanism. We believe that most of the applications should checkpoint in a more moderate rate. However, by doing this, our propose was to evaluate checkpointing scalability and compensate the reduced number of machines available for the experiment.

The application was submitted twice: one with the normal checkpointing and one with the Unified Checkpoint mechanism. Each submission used 16 replicas to perform $2.88 \times 10^{12}$ iterations. This is a experimental value we have adopted so the implementation period exceeds 24 hours, enabling us to observe the fault tolerance mechanisms during the execution of a long application. During execution, all fault tolerance mechanisms - retrying, replication and checkpointing - remained active. The delay ratio adopted to replace replicas was 0.5. With the normal checkpointing, the running time was 63 hours and 30 minutes. With the Unified Checkpoint, the running time was 40 hours and 42 minutes. That is, in this example, we save

Copyright © 2000 John Wiley & Sons, Ltd.



more than 20 hours of execution. Only four replica replacement actions were performed during the Unified Checkpoint execution, due to the small heterogeneity of the cluster.

For monitoring the memory consumption of the main container, we used the JConsole tool [24]. This tool connects to an instance of the Java virtual machine, allowing the view of several information as the memory size (heap), the number of classes loaded into memory, and the number of running threads.

In normal checkpointing, the average memory consumption after the submission was about 17 megabytes, with a peak of 30 megabytes. With Unified Checkpoint, the average was about 20 megabytes and the peak was 34 megabytes. The difference, therefore, was only 4 of megabytes, due to an increase in the number of objects loaded into memory in Unified Checkpoint solution.

The CPU usage and total memory consumption was low (0,8% and 7,0% respectively, considering *orlandia* hardware specification) and remains the same in both checkpointing solutions. These results show an small increase of 17.6% in memory consumption and low CPU usage demonstrating the technical feasibility of the Unified Checkpoint mechanism.

## 7. CONCLUSIONS

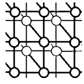
Grid computing middleware hides the complexity related to distribution and heterogeneity. It seeks to address issues such as management and allocation of distributed resources, dynamic task scheduling, fault tolerance, support for high scalability and heterogeneity of software and hardware components, protection, and security.

The mobile agents paradigm is suitable for dealing with the complexity of building the grid software infrastructure due to its intrinsic characteristics such as cooperation, autonomy, heterogeneity, reactivity and mobility. In this work, we presented the Unified Checkpoint mechanism, which combines dynamic task replication, replica substitution and checkpointing to provide fault tolerance for sequential and parametric applications. We used the MAG middleware as the basis for implementing these mechanisms. This middleware benefits from the mobile agent paradigm to encapsulate the applications submitted to the grid into mobile agents that control the applications life cycle and exchange messages to coordinate fault tolerance actions.

The improvements made on MAG are towards an adaptive middleware, capable of altering its behavior accordingly to environment changes. The results showed that, within the observed parameters, our solution helps to reduce the applications execution time further than the previous model, in which the checkpointing mechanism and replication were not integrated. The results were favorable to Unified Checkpoint, with a major discrepancy noted in the submission of long tasks in large clusters. Through the experiments, we submitted an application in a real cluster of 16 machines managed by the middleware InteGrade/MAG and we also compared our solution to the previous one. The results showed that our solution results in a higher memory consumption, but the increase is not significant.

Currently, we are still investigating other self-optimization and adaptive mechanisms to add to our feedback system. We are measuring the benefits of increasing or decreasing the number of replicas dynamically according to three factors: failure rate of the execution environment, number of free resources, and amount of tasks to be scheduled. We are also investigating the

Copyright © 2000 John Wiley & Sons, Ltd.





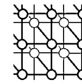
impact of changing the checkpointing interval according to the failure rate and the size of the checkpoints to optimize application completion time.

**REFERENCES**


1. Steere DC, Goel A, Gruenberg J, McNamee D, Pu C, Walpole J. A feedback-driven proportion allocator for real-rate scheduling. *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation.* USENIX Association, New Orleans, Louisiana, 1999. 145–158.
2. Goel A, Steere D, Pu C, Walpole J. Adaptive resource management via modular feedback control. *Tech Report.* Oregon Graduate Institute, Department of Computer Science and Engineering, 1999.
3. Pham VA, Karmouch A. *Mobile Software Agents: An Overview.* Communications Magazine, IEEE, 1998. **36**(7):26–37.
4. Barbosa RM, Goldman A, Kon F. A study of mobile agents liveness properties on MobiGrid. In 2nd International Workshop on MATA, Montreal, Canada, 2005. 17–19.
5. Barbosa RM, Goldman A. Framework for Mobile Agents on Computer Grid Environments. In First International Workshop on MATA, 2004. 147–157
6. Lopes RF, Silva FJS, Souza BB. MAG: A Mobile Agent based Computational Grid Platform. Proceedings of CCGrid'05. LNCS Series, Springer-Verlag, Beiking, 2005.
7. Goldchleger A, Kon F, Goldman A, Finger M, Bezerra GC. InteGrade: object-oriented grid middleware leveraging the idle computing power of desktop machines. *Concurrency - Practice and Experience*, 2004. **16**(5):449–459.
8. Camargo RY de, Cerqueira R, Kon F. Strategies for Storage of Checkpointing Data Using Non-Dedicated Repositories on Grid Systems. ACM/IFIP/USENIX 3rd International Workshop on Middleware for Grid Computing, Grenoble, France, 2005.
9. Camargo RY de, Cerqueira R, Kon F. Strategies for Checkpoint Storage on Opportunistic Grids. IEEE Distributed Systems Online, 2006.
10. Cirne W, Brasileiro F, Andrade N, Costa LB, Andrade A, Novaes R, Mowbray M. Labs of the world, unite!!!. Journal of Grid Computing, Springer, 2006. **4**():225–246.
11. Santos-Neto E, Cirne W, Brasileiro F, Lima A. Exploiting Replication and Data Reuse to Efficiently Schedule Data-Intensive Applications on Grids. Job Scheduling Strategies for Parallel Processing. LNCS Series, Springer, Berlin/Heidelberg, 2005.
12. Hwang S, Kesselman C. A flexible framework for fault tolerance in the grid. Journal of Grid Computing, Springer, 2003. **1**():251–272.
13. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the Condor experience. Concurrency and Computation: Practice and Experience, 2005. **17**(2-4):323–356.
14. O'Brien PD, Nicol RC. FIPA - towards a standard for software agents. BT Technology Journal, July, 1998. **16**(3):51–59.
15. Cao J, Jarvis SA, Saini S, Kerbyson DJ, Nudd GR. ARMS: an Agent-based Resource Management System for Grid Computing. Scientific Programming (Special Issue on Grid Computing), 2002. **10**(2):135–148.
16. Beguelin A, Seligman E, Stephan P. Application level fault tolerance in heterogeneous networks of workstations. Journal of Parallel and Distributed Computing, 1997. **43**():147–155.
17. Buyya R, Murshed MM. GridSim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing CoRR, 2002. http://arxiv.org/abs/cs.DC/0203019
18. Plank JS, Elwasif WR. Experimental assessment of workstation failures and their impact on checkpointing systems. 28th International Symposium on Fault-Tolerant Computing, Munich, June, 1998. 48–57.
19. Truyen E, Robben B, Vanhaute B, Coninx T, Joosen W, Verbaeten P. Portable support for transparent thread migration in Java. In ASA/MA, LNCS Series, Springer-Verlag, September, 2000. 29–43.
20. Fukuda M, Smith D. UWAgents: a mobile agent system optimized for grid computing. International Conference on Grid and Applications - CGA'06, Las Vegas, 2006. 107-113.
21. Martino B, Rana OF. Grid performance and resource management using mobile agents. Performance analysis and grid computing, Kluwer Academic Publishers, Norwell, MA, USA, 2004. 251–263.
22. Loke SW. Towards data-parallel skeletons for grid computing: an itinerant mobile agent approach. Proceedings of the CCGrid 2003. 651–652.
23. InteGrade. http://www.integrade.org.br [30 Oct, 2010].


Copyright © 2000 John Wiley & Sons, Ltd.

*Prepared using* cpeauth.cls




24. , Mandy Chung. Using JConsole to Monitor Applications.
http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html [30 Oct, 2010].
25. Portable Operating System Interface.
http://standards.ieee.org/regauth/posix/index.html [30 Oct, 2010].
26. JADE - Java Agent DEvelopment framework.
http://www.jade.tilab.com [27 Feb, 2010].
27. JADE Administrator's Guide. Bellifemine F, Caire G, Trucco T, Rimassa G, Mungenast R.
http://jade.tilab.com/doc/administratorsguide.pdf [30 Oct, 2010].
28. SPECfp 2000 - Processor Benchmark.
http://www.spec.org/cpu2000/results/cfp2000.html [30 Oct, 2010].
29. SETI@home: Search for Extraterrestrial Intelligence at home.
http://setiathome.ssl.berkeley.edu/ [01 March 2010].
30. Aglets Software Development Kit.
http://www.trl.ibm.com/aglets/ [27 Feb, 2010].
31. BOINC: The Berkeley Open Infrastructure for Network Computing.
http://boinc.berkeley.edu/ [27 Feb, 2010].
32. The Great Internet Mersenne Prime Search.
http://www.mersenne.org [27 Feb, 2010].
33. Einstein@home - Catch a Wave from Space.
http://einstein.phys.uwm.edu [30 Oct, 2010].
34. Do it yourself climate prediction.
http://www.climateprediction.net [27 Feb, 2010].


Copyright © 2000 John Wiley & Sons, Ltd.

*Prepared using* cpeauth.cls