Fault Tolerance features in GRID superscalar

Raül Sirvent, Rosa M. Badia and Jesús Labarta
Barcelona Supercomputing Center
Barcelona, Spain
Email: [raul.sirvent, rosa.m.badia, jesus.labarta]@bsc.es

***Abstract*—Regarding Fault Tolerance, several techniques and strategies are well-known and commonly used. However, not all of these techniques can be applied to all the possible scenarios. In this paper we discuss how GRID superscalar, a programming model which tries to ease the programming of Grid applications, can benefit from Fault Tolerance. We describe the new mechanisms implemented in the runtime in order to deal with failures: automatic drop of machines, soft and hard timeouts for running tasks, asynchronous results' transfer during an application's execution, retry of operations in the library and avoiding situations that previously made the application to stop. We specify the benefits and drawbacks of these implemented mechanisms, and we describe the execution of a Monte Carlo algorithm which allows us to prove their usefulness, specially in long executions. We also propose as future work other mechanisms to increase even more the tolerance to failures of the system.**

I. Introduction

It is well known that the Grid is a distributed collection of services and resources which can be used in a remote fashion. But it is also important to remember that, as a distributed system, the Grid is an environment prone to failures. A Grid may be formed by thousands of machines or clusters, geographically distributed, with different administrators, so, many abnormal behaviors can arise: network failures, machine failures, machines leaving the Grid, and so on. A Grid component should be aware of such situations, in order to react properly, thus not having to stop because something happened in a machine. In this scenario is where Fault Tolerance (FT) techniques grow in importance.

When reading literature about Fault Tolerance we can see that it can be applied to different areas such as architecture design [1], hardware [2] or software [3] [4]. A failure can be defined not only as a software or hardware crash, but also as a system not accomplishing some requirements (i.e. a performance degradation). Failures (hardware or software) are usually classified in three categories:

- Transient failures: They appear once, and then disappear.
- Intermittent failures: They appear and disappear randomly (without following a pattern).
- Permanent failures: Whenever they appear the only solution is to repair or replace the faulty component.

In order to overcome these failures, the most common strategies used are based on redundancy. This redundancy can also be categorized in three classes:

- *Information redundancy:* Add information for detecting or recovering failures.
- *Time redundancy:* Retry a certain operation which has failed.
- *Physical redundancy:* Adding extra hardware or software to the system.

The strategies are also known as *backward recovery*, when the system returns to a previous correct state, and *forward recovery*, when the system is brought to a correct state. As an example, a log or checkpoint mechanism can be identified as an information redundancy and backward recovery mechanism, and a retry mechanism would be a time redundancy and forward recovery mechanism.

GRID superscalar [5], as a Grid component, must be aware of failures. In this paper we discuss which are the mechanisms suitable for GRID superscalar in order to deal with failures. The runtime already has a checkpointing mechanism (backward recovery) in order to avoid repeating the whole computation whenever a failure appears: the application is stopped, the error can be corrected manually, and then the computation can continue from the checkpoint file. The main goal of this work is to allow an application running with GRID superscalar to keep running despite of the possible failures that can arise during the execution. It is clear that the checkpoint mechanism and the new Fault Tolerance mechanisms added are complimentary (in case of an unrecoverable failure, a user may restart the execution from the checkpoint).

Section II will introduce the GRID superscalar programming environment. Then section III will detail more specifically the mechanisms implemented in the runtime for making it fault tolerant, combining forward and backward recovery strategies with information and time redundancy. Section IV presents the results of a designed test case which allows us to demonstrate the usage of these new mechanisms. And section V will draw some conclusions and future work we envision.

II. GRID superscalar overview

As a programming model, GRID superscalar is focused on easing the programming of Grid applications. It is clear that the easiest way of programming for a user is with the desired programming language, in which the user is already familiar with, and in a sequential fashion, without using complicated parallel schemes where the user must control syncronizations, message passings, and so on. GRID superscalar achieves this by providing bindings to different programming languages (currently C/C++, Perl, Java and Shell script), and a runtime environment which automatically executes in parallel the user-defined functions that do not have data dependencies between

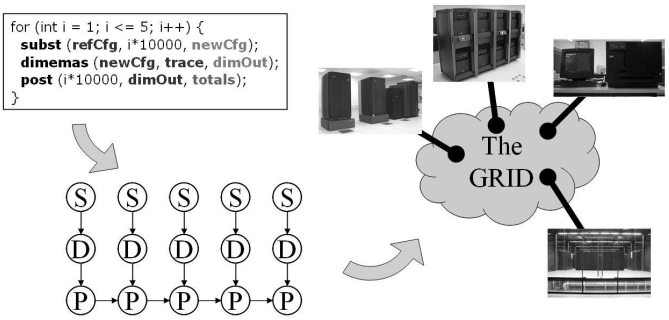

Fig. 1. GRID superscalar in a nutshell

them. From a source code, GRID superscalar builds internally a workflow with the existing data dependencies between functions, as shown in figure 1, and from that workflow the tasks without dependencies are considered to be run on the Grid. GRID superscalar is not only a programming model, but also a set of tools that allows users to easily *gridify* an application.

In order to program an application with GRID superscalar, a developer must provide a main program, the code of the functions in that specific main program to be executed in the Grid, and an IDL file, which describes the interface of these functions (the type of the parameters, and the direction of these parameters). There is a small set of calls that must be added in the main program: *GS_On* for starting the runtime, and *GS_Off* for stopping it. Calls for handling local files (*GS_Open/GS_Close, GS_FOpen/GS_FClose*), as the file is the data unit considered for detecting the dependencies between the functions. Also more advanced, but optional, primitives are provided, such as *GS_Barrier* to wait for all generated tasks to finish, and *GS_Speculative_End* to easily create optimization-like algorithms.

A tool named *Deployment Center* allows not only to graphically specify and check the Grid configuration, but also to deploy (send and compile) locally and remotely the code involved in the developer's project. This deployment step is assisted by *gsstubgen*, another tool in charge of generating stubs and skeletons needed for allowing the runtime calls, and *gsbuilder*, responsible for compiling the generated code. This process creates a local client binary and several remote server binaries, which leaves a master-worker paradigm application ready to run.

When the user invokes the local binary (the master or client) the runtime comes into action. It starts building a Directed Acyclic Graph containing the data dependencies between the tasks generated until that moment, and at the same time starts submitting the tasks which are ready (with no dependencies) to the available machines, thus achieving its parallel execution. In order to decide which is the most suitable host for a task, an estimation of the task's execution time provided by the user is considered, as well as the time that will be spent to transfer all input files required.

The runtime is also in charge of transferring the files needed by a task to the selected host, submitting the task, and after completion, transfer the results back to the master. These operations are more prone to failures, as they are remote and executed by a middleware that provides the basic Grid services to the runtime. As a consequence, our main goal is to achieve the correct execution of these basic interactions with the Grid middleware despite the possible failures that can arise.

The input and output files related to a task are kept in the worker until the end of the execution to try to exploit the locality of files, thus saving file transfers. This means that the results of a task can be used when the task finishes, without having to wait for the transfer to the master to end. When a task has finished, the data dependence graph is updated (this can generate new ready tasks) and the resource becomes available for executing a new task. At the end of the execution, the remote files are cleaned up, and the remaining results are transferred back to the master, leaving everything as if it had been a local execution of the application. Other techniques are used in the runtime, such as file renaming to erase false data dependencies, disk sharing to make GRID superscalar aware of data replicas or real shared file systems, checkpointing in order to restart the computation from where it stopped because of a big failure (detailed in section II-A), and ClassAds [6] constraints specification to filter the resources in the Grid. Also dynamic host reconfiguration is offered to add or remove machines to the computation.

For monitoring the execution, the GRID superscalar Monitor can be used. This tool is very useful to visualize the task dependence graph in order to investigate why the application does not reach the desired parallelism. It also shows the status of the tasks: if a task is running it states the machine where the task is running, and when a task is done, it still holds the information about where it has been run, thus providing a graphical way of determining which hosts are executing more tasks.

This programming model has been adapted to several environments, currently: Globus (which can work with versions 2 and 4 of the Globus Toolkit [7]), ssh/scp and Ninf-G [8].

Regarding the ssh adaptation of the programming model, one of the objectives was to overcome the overhead detected in some Grid middlewares when submitting small granularity jobs. Also if a user wants to work with GRID superscalar inside a cluster it makes no sense in introducing the overhead of calling to a Grid middleware in order to operate between the different nodes, because all the resources are local. Inside a cluster there is no need of encrypted communication, so an easier task notification mechanism can be used, based on TCP/IP sockets.

The Ninf-G adaptation offered several advantages for GRID superscalar when using a Grid middleware. Ninf-G has an advanced file transfer protocol and the possibility of creating persistent workers. Ninf-G is a GridRPC implementation, thus provides a simpler interface, in contrast to Globus, where the job submission is based on building the corresponding RSL.

The current ongoing developments of the programming model have a different general approach for achieving the

parallelization of the code. Instead of using generation of intermediate code from the IDL file, the new version is based in code annotations and using a source to source compiler. It offers new features such as full support for scalar variables, support for multidimensional arrays and structs only containing scalars, client side worker threads and tracing for post-mortem analysis.

A. Checkpoint mechanism

The checkpoint mechanism included in GRID superscalar was an early step to achieve tolerance to failures. It is classified as a backward recovery strategy (we want to go back to a correct state) and with information redundancy (we need to store extra information about the tasks finished correctly). The main idea in its design was to stop the application when an error was detected, but saving the work performed until that failure in order to avoid repeating it. This allows the user to correct the error (if his/her interaction is needed for that purpose) and to restart the application from the point where it stopped. This mechanism did not distinguish between recoverable or unrecoverable failures, and this is one of the things that we want to overcome in this work.

Task errors in GRID superscalar can be detected in two levels: at a middleware level or at a task level. The Grid middleware can notify to the master when something related to itself is failing for an executing task (i.e. a Grid service not working correctly). At a task level the error can be risen by two main causes. The first one is a signal received by a worker (caused by a segmentation fault, division by zero, or because the process has been notified with a SIGTERM that it must stop). The second one is related to the application: the user detects a wrong result in a task and wants to notify that to GRID superscalar.

GRID superscalar's checkpoint works at a inter-task level. This means that the task's status is only saved if its execution has finished correctly. Tasks which may fail during their execution will have to be restarted from the beginning. This can be a problem when the execution time of a task is big. In these cases, we recommend the programmer to use an application level checkpoint, tailored to his/her needs, or another low level checkpoint mechanism already available (like the ones described in [9]).

In our strategy, we assume that we can only checkpoint a task if all its predecessor tasks in a sequential order have finished. This sequential order is determined by the sequential execution of the application's main program. This assumption reduces the complexity of writing a checkpoint because we do not need to write all the data structures in memory to the disk, and it also reduces the number of files that we must store in order to restart the application correctly. With the renaming technique applied in the runtime, a file can have a lot of different versions from itself, and in the worst case, we would need to store all of them in order to be able to restart from a specific point. This mechanism restores what we call the *sequential consistency* of the application: it stops the application as if a sequential application was stopped. The drawback is that some tasks may have finished their execution, but we are not able to checkpoint them because some of their predecessors have not finished. However, this side effect can be lowered by influencing the scheduling decisions in the runtime (giving more priority to tasks with a smaller task number, thus, prior in the tasks generation). In addition, once a failure is detected, the runtime waits for the running tasks that have the possibility of being checkpointed. The tasks which cannot be saved are cancelled to avoid wasting time with them. We can see an example of this behavior in figure 2. Tasks 0 and 1 have been previously checkpointed, tasks 2, 3 and 5 are running, and tasks 4 and 6 are pending. Task 3 finishes its execution but it cannot be checkpointed because it has predecessors still running (task 2). After that we receive a notification about a failure in task 2, which means that we won't be able to checkpoint tasks with a higher sequence number. The immediate decision is to discard task 2 even though it has finished, and cancel task 5 because we will not be able to checkpoint it even if it executes correctly.

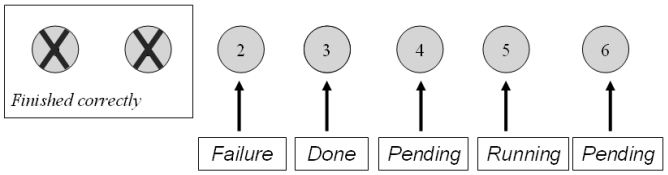

Fig. 2. Checkpoint example

So, in order to maintain a task's state when its execution is correct, we need to store the task number (if the previous task has finished), and know the current versions of all the files in that given moment to save them in case of error. More precisely, we need to save all results generated by any previous task which may be considered a final result (this means, it must be the last valid version of the file). To these files we need also to add all the files opened in the main program with GS_Open and GS_FOpen with read mode, because, when the user restarts the execution, the instructions after an open must read the same values they read during the first execution. The same happens with the output scalar values of a task. As we are not going to execute the task again, we need to store the results this task has generated. This is very important, because the execution flow in the main program must be reproduced exactly as the previous execution. If a decision in this execution flow depends on an intermediate value, the decision must be the same in the restart.

One of the features of this checkpoint mechanism is that the user does not need to interact with the runtime in order to enable it: it is completely transparent to the application. In order to restart an execution which has failed, the user only needs to re-run his/her main program. If a checkpoint file is found, the runtime will start automatically from the stored point. And if there is no checkpoint file, the execution will start from the beginning.

III. Adding Fault Tolerance to GRID superscalar

Our main approach in order to add Fault Tolerance mechanisms to GRID superscalar is that we want the application to keep running even when failures arise. Following the classifications made in section I, when we consider software failures we can tackle the three types of failures using time redundancy (retrying the operation), as it may be very difficult to recover from a Grid middleware error with information redundancy at application level. Also using physical redundancy (submitting a task more than a single time at the same moment) may cause a high overload in the system. When focusing on hardware failures the situation is even less in control of the application, as the user cannot have a deep control of the machines that belong to the Grid (and he/she is probably not interested in having such a control). Transient and intermittent failures can be handled by retrying the operation which has failed, but permanent failures can only be treated by removing the machine which is failing from the pool of machines available to the computation. Anyway, whenever the permanent failure is solved, a user may add again the machine to the computation dynamically at run time.

Next sections will describe the specific implementations in order to overcome the failures detected when running applications with GRID superscalar.

A. Automatic drop of machines

When a machine is not able to execute jobs because of a failure, GRID superscalar must be aware of that and adapt its behavior to the new circumstances. This situation can occur frequently, if we see the Grid as a large collection of heterogeneous machines. Previously, a final user was able to drop manually a machine from the computation with the dynamic host reconfiguration script, which reads again the GRID superscalar configuration file that specifies the hosts that can run jobs for a particular application. This solved the situation where a machine is giving a poor performance, and thus slowing the overall execution. The problem was that a remote failure in the execution made the master to stop, notifying that to the user. Now, at run time, whenever a job execution fails, the Grid services are checked (i.e. Globus) and if they are running correctly the operation is retried a maximum of three times. If the check fails or the retries reach the retry limit, the machine will not be used to execute other tasks.

In order to drop a machine, the runtime waits for other jobs running there, because it may happen when only the Grid middleware fails that new submissions do not work, but previous running jobs keep executing and sending notifications to the master, or even the service for transferring files can keep running. If the failure is a general failure of the remote machine, running tasks will be removed with the timeout mechanism described in section III-B. Because of the policy of exploiting file locality implemented in GRID superscalar, it could be possible that some result files are only available in the machine that must be dropped. So, before dropping the machine the runtime checks if some files are only available in the working disk of that machine, and tries to transfer them to the master. If the file transfers also fail, the only possible solution is to restore the sequential consistency, stopping all the remote tasks, and resuming from the last checkpointed task. This will be also done automatically at run time by the library without any additional effort from the user and allowing the execution to continue.

When a machine is dropped, the machine information is kept in the runtime for future use (i.e. the user fixes the problem and wants to add the machine again to the computation), but no job can be submitted to it.

B. Soft and hard timeouts for remote tasks

Several kinds of failures might appear when a machine runs a job. In any of the possible failures, the key is to be able to determine that something is happening, and correct the wrong behavior. We have taken the task time estimations given by the user as a reference to do so. As it has been explained in section II, the user can specify the estimated time that a task is going to need to be run. From that time, we define two upper bounds: the timeout factor and the resubmit factor.

The timeout factor determines the time when the master is going to ask for the status of a remote job. It is calculated multiplying the estimated execution time for a task, and the specified timeout factor (its value defaults to 1.25). For instance, if a user has specified a estimated time for a task of 100 seconds, a timeout factor of 1.25 means that for this task, when the execution time reaches 125 seconds, the runtime is going to check that the task is still running. If the Grid middleware responds correctly, the checks will continue every 30 seconds (to avoid overloading the machines, and in correspondence to Globus poll job manager interval) until we reach the time specified by the resubmit factor. If the task does not respond, or the status is not running, the task is resubmitted in the same machine. This is done because a transient failure may have affected the task, but the machine may still be able to run new jobs. When a machine reaches 3 resubmits of tasks, it will be dropped as described in section III-A, and the task will be resubmitted into a new resource.

The resubmit factor is like a hard limit for the task execution time. It specifies the time when the master is going to kill the task, because the user believes that the execution has gone abnormal. This will avoid the master waiting for a specific task for a long time because of a performance degradation in a specific machine, thus delaying the whole execution process, and also avoids the master to hang up when a task does so. The resubmit factor is calculated as the timeout factor: multiplying the estimated time of the task and the resubmit factor. Its value defaults to 2, so in an estimated time of 100 seconds, the hard limit will be reached at 200 seconds. Whenever this limit is reached for a task, the task is killed and resubmitted in the same machine. The reason for this is because the failure may be caused not only by a software component (the task) but also by the hardware. So we first resubmit in the same machine, believing that the failure comes from a temporary failure in the hardware (i.e. a performance degradation due some overload

period of the machine), a maximum of three times. When a machine has three resubmits, it may have a permanent failure in terms of performance, which allows us to submit jobs but they execute very slow, so we resubmit the task into a new machine.

As it has been explained, both parameters have a default value. However, the timeout and resubmit factors can be specified by the user with the environment variables GS_TIMEOUT_FACTOR and GS_RESUBMIT_FACTOR, always taking into account that GS_TIMEOUT_FACTOR must be bigger than 1, and that GS_TIMEOUT_FACTOR must be smaller than GS_RESUBMIT_FACTOR. This allows the user to tune the behavior of these two mechanisms.

C. Asynchronous transfers of results for each task

The checkpoint mechanism previously available in GRID superscalar is no more valid when we expect machines to be completely unavailable: the results of each task were only stored in the worker that run the task in order to exploit the data locality for upcoming tasks. Thus, if a task is checkpointed, but its results are in a machine that we cannot access, we reach an inconsistent state. This is also important when a machine is dropped, because the files only available in that machine must be recovered in order to reach a consistent state in the checkpoint. Of course, if a file corresponding to a task that has been checkpointed is no more available, the checkpoint can be undone, but we foresee that this will not be a good solution, as the checkpoint undo process could undo all the checkpoints done util that moment, thus not having checkpoint at all.

In order to improve the checkpoint mechanism to avoid these inconsistent states, we transfer the results of each task (once the task has finished) to the master, but leaving a copy in the worker aswell, to keep exploiting the locality of data. This transfer of results is done in a background process during the execution, so hidding the time taken for transferring the files to the master and not delaying the whole execution. So, with this new mechanism, a task will be checkpointed when the results are available in the master. This has some advantages and some drawbacks. We increase the availability of a file, so if a remote machine must be dropped, we will still have a copy of the result files in that machine in the master, thus avoiding to have to restart at run time from the checkpoint file, as described in section III-A. However, if the transfer of results is not completed before the failure appears, the automatic recover mechanism from the checkpoint file is still needed. Another key point is that once a task is checkpointed, now we can ensure in all the cases that the task checkpoint is not going to be undone, so the task will not be computed again. And, as a collateral benefit, the fact of transferring task results to the master during the execution makes that, at the end of the whole execution, the postprocess needed to retrieve final results to the master is very small or inexistent. In brief, we are hidding the postprocess time with the execution time, speeding up the whole execution specially when the main program has lots of result files with a single version for that file (every time a file is renamed in the runtime with the renaming mechanisms, a new version of that file is created). We also identified two drawbacks: a bigger usage of the master's disk and more transfers to the master, because previously the different versions of a file were stored in the remote workers, and only the final version was transferred to the master. Nevertheless, the benefits are bigger than the drawbacks identified.

D. Retry of operations inside the library

As we have seen in section III-A, a machine can be dropped from the computation when the Grid services available are not working. More precisely in the job submission related to the postprocess, before dropping the machine, several actions are considered in order to submit a job. The objective of the job submission related to the postprocess is to cleanup remote old versions of result files as well as input files, and retrieve the latest version of result files. In GRID superscalar several machines can share a working directory, so when a job submission fails for a specific machine, a different machine with the same disk can be considered for submitting this job related to the postprocess. For example, machine A and B work with the same working disk named Disk1. If we are not able to submit a job to A in order to cleanup and retrieve files, we can try the same thing with B, obtaining the same final result: Disk1 has been cleaned up. Therefore, the retries implemented are not only related to machines, but also to disks. The same is done when asynchronously retrieving result files of a task (mechanism described in section III-C).

Inside the master library, there are other operations that can fail, rather than Grid middleware operations. These operations are system calls. Our main objective when implementing retries in system calls inside the library is to avoid transient failures which may occur in the master machine (I/O errors, etc...). With these retries, we can save the master from having to stop because a transient failure has appeared in its own machine. As system calls are very fast in response, in contrast to remote calls to the Grid middleware, the retries have been delayed with 1 second each, and increased to 5. This is translated to a period of 5 seconds retrying a system call whenever it fails, which is not a big overhead to the master, and it is enough time to consider the failure as permanent.

E. Avoid situations that make the master stop

As we have seen in previous sections, the main objective of the Fault Tolerance mechanisms implemented is to avoid the master to stop the computation. This can happen not only because Grid middleware failures, but also because of system calls inside the library. However, there is still a cause of failure inside the master library, and is related to the size of strings. We have made a big revision to the library code, and we have removed all the avoidable situations that made the master to stop. One example of this is in the sockets mechanism implemented for receiving messages from tasks. This mechanism got a fixed maximum message size, which could be modified by the environment variable

TABLE I
BEHAVIOR EXAMPLES

Failure	Behavior
The middleware notifies a task failure	Check Grid services. Resubmit in the same or a new machine
A task reaches the soft timeout	Ask for the status every 30 sec. If no answer, check Grid services and resubmit in the same or a new machine
A task reaches the hard timeout	Cancel the task. Resubmit in the same or a new machine
A system call fails in the runtime	Retry 5 times with 1 second delay
A check of Grid services fails	Drop machine
More than 3 retries in a machine	Drop machine

GS_MAXMSGSIZE. Previously, if a message was bigger than the maximum size specified, the master stopped with an error. Now, the master resizes dinamically the buffers for receiving messages, and reads repeatedly from the sockets to retrieve the whole information. The same happens for other pre-defined parameters, as GS_MAXPATH (maximum size of a path), GS_GENLENGTH (maximum size of a scalar parameter), and so on.

Table I details some examples where the mechanisms described are used to detect and overcome a failure.

IV. EXPERIMENTAL RESULTS

All the mechanisms described in previous sections have been implemented and tested in current distribution of GRID superscalar. For this paper we have designed a long test case as a proof of concept about the new features available. We must clarify from the beginning that the main goal of a test when talking about Fault Tolerance is to keep the application running despite the failures that can arise in the different machines that form our Grid. We are not expected to give a better performance than with a error-free environment, but we will be able to see how these errors affect to the expected performance.

We have implemented a Monte Carlo algorithm, composed of 960 simulations, where every simulation takes 1 hour time to run. This means that the total time for executing this algorithm sequentially can be roughly estimated in 960 hours (40 days). Our Grid environment for running this experiment is composed of four PCs, with a Intel Pentium 4 CPU 3.00GHz processor with Hyper-Threading technology and 1 GB of RAM. These computers have Globus installed, as well as the latest GRID superscalar distribution. We run 2 simulations at the same time in a single processor, thus the estimated execution time in our Grid is 120 hours (5 days), because they can be run in parallel as no data dependencies exist between the tasks.

Another important part of the experiment is the design of the failures. We have simulated four different kinds of failures.

TABLE II
SUMMARY OF FAILURES

Day	Machine	Type	Expected Job Failures	Expected Drops
Day 1	bscgrid02	Kill Workers	12	3
Day 2	bscgrid03	Kill Service	12	3
Day 3	bscgrid04	Performance	8	2
Day 4	bscgrid01	Stop Service	10	5

The first one was to kill the running workers in a remote machine. This could happen if a machine administrator or a local resource manager kills the processes for any reason, or even also if the machine is shut down. The second type was causing a failure in the Grid service for job execution (the processes that control the worker) by killing it at run time. This may also happen in a similar scenario as the first type of failure. The third type pretends to simulate a performance degradation in a machine. We achieved this by changing the simulation that must be done by a longer one, thus making execution time bigger than expected. The last type of failure is to stop the job execution Grid service in order to make new requests to the machine fail. These failures have been implemented with scripts that are submitted at the same time we submit the application, but keep slept until the moment they rise a failure. We have focused the failures in the job execution service because it controls also the job transfer service. Any failures in file transfers would be detected and notified by the job execution service, so, they are already covered by the middleware.

As the test case is big enough to do it, we have distributed the types of failures defined in different days: first day one machine fails by killing the workers, second day another machine fails by killing the job execution service, the third day we simulated performance degradations, and a fourth day we stopped the job execution service. The first and the second day we caused an error every 4 hours, rising 6 failures per day. The third day we changed the simulator every 4 hours for a time period of 2 hours (4 failures per day). And the fourth day we have enabled or disabled the job submission service every 2 hours, letting this service half of the time avaiable. It is also important to mention that we have set the soft timeout to 1.05 and the hard timeout to 1.10 of the estimated execution time, in order to lose the minimum time to detect a failure. We can see a summary of the failures design in table II.

In the previous section we have described the mechanism of dropping machines from the computation when problems arise. If we cause failures, a machine can be dropped from the computation, but if all the machines in our Grid are dropped, the execution will have to stop. In order to avoid this we have submitted the master process with a script that reads again the configuration file after a machine drop is expected, This will add again to the computation the machines that where dropped before.

The results of the execution are described in table III. The

TABLE III
RESULTS

Machine	Completed	Failed	Jobs per Day	Drops
bscgrid01	190	26	42, 46, 48, 24, 40, 30	7
bscgrid02	255	12	33, 48, 48, 48, 48, 30	3
bscgrid03	255	12	48, 33, 48, 48, 48, 30	3
bscgrid04	260	8	48, 48, 38, 48, 48, 30	2

total execution time has been 5 days 15 hours 47 mimutes and 39 seconds. A first thing to highlight is that the ideal number of jobs per day executed in a machine is 48. When no errors are introduced, all the machines but bscgrid01 accomplish this objective. Looking at the execution logs, we can see that in bscgrid01 executions we have had hard timeouts for some running tasks due to performance degradation. This is caused because our soft and hard timeouts are very narrow to the real execution time, and the usage of bscgrid01 is higher compared to the rest of bscgrid machines (it has usually more people working in it than the rest). This has caused 16 extra retries and 2 extra drops which were not initially planned. We could think that 16 errors should cause 4 drops, because a machine is only retried 3 times before dropping it. This behavior is explained because when the configuration file is read again the retry counter for a machine is set to 0, causing that we only have an extra drop if 4 errors occur close in time (this happens for bscgrid01 the first and the fifth day).

Because of the overhead of the middleware and file transfers, the last 2 tasks in a day finish their execution in the next day of the computation. We have added them to the day where they started for simplicity when trying to understand the results. We can see that in bscgrid02 (the machine failing the first day), there have been 12 tasks that have been resubmitted. The failure is detected when the soft timeout is reached and the runtime starts asking about the status of the task. As the contact cannot be established, the task is resubmitted immediately. Once 4 tasks fail, the machine is dropped from the computation, having a total of 3 drops for bscgrid02 (as we recover it once is dropped). In bscgrid03 (second day of failures) we have a similar scenario: 12 tasks resubmitted and causing 3 drops of machine. Anyway, in this case the errors are detected earlier: once the worker is killed, the middleware sends a notification to the runtime and resubmits the job immediately. The performance degradation in bscgrid04 makes the task to reach the hard timeout: the runtime kills the running task, and resubmits it. The test shows that 8 tasks had this problem, and the machine has been dropped 2 times. When stopping the job execution service in bscgrid01 the fourth day, the failure is detected when a new job is going to be submitted. The runtime tries to submit 10 jobs that fail when the service is not available (causing 5 drops of the machine). The errors detected in previous days and the two extra drops have been explained before.

The total execution time was expected to be 5.25 hours slower, as we have caused in purpose failures for 42 jobs during the first 4 days. In the end, the execution has been delayed approximatedly 15.5 hours. This is due not only to bscgrid01 unexpected failures (16), but also to a non suitable drop of machines for this particular configuration. Except for the job submission service failure, in the rest of cases the drop is caused when a fourth task needs to be resubmitted, thus in every drop we have 1 running task instead of the two possible. This makes the jobs per day value decrease in one job in these drop cases. Also the delays in re-adding machines that have been dropped add more time to the total. It is important to mention that despite all the failures registered during the whole execution (a total of 58), the application finished successfully and the results were correct.

V. CONCLUSIONS AND FUTURE WORK

In this paper we have seen how a well-known Fault Tolerance mechanism (retry) can be added to a Grid programming tool in order to overcome the possible failures which may arise during the execution of an application, thus adding forward recovery strategies to the previous backward recovery strategy available (the checkpoint mechanism). We have discussed how automatic drop of machines can help to skip permanent hardware failures. We also have implemented timeout and retry mechanisms for tasks in order to detect and solve any kind of failures (not only software, but also hardware) while a job is running in a remote machine. The checkpoint mechanism has been improved in order to increase its reliability, and the postprocess time needed when a GRID superscalar application finishes has been reduced. The retry mechanism has been also focused to the possible transient failures in the master machine whenever using system calls, or other situations which previously made the master to stop. We have presented experimental results which allow us to confirm that our mechanisms work correctly in a simulated faulty environment. More precisely, an algorithm that was planned to run ideally in 5 days in the Grid has been run half day slower, but dealing and overcoming all the errors detected during the execution. In addition, some of these errors appeared spontaneously, without being initially planned.

We believe that this work is a good step towards achieving Fault Tolerance in GRID superscalar. However, it is clear that more work must be done in the master part of the system to avoid having a single point of failure. We plan to implement mechanisms similar to the one described in [10], to be able to replicate the master, and control the master behavior with operation logs. We need also to investigate the possible overhead of implementing physical software redundancy (submitting several times the same task at the same moment), as well as the benefits obtained.

ACKNOWLEDGMENT

This work has been partially supported by the CoreGRID Network of Excellence (contract IST-2002-004265) and the Ministry of Science and Technology of Spain (contract TIN2007-60625).

REFERENCES

[1] M. Hecht, J. Agron. *A distributed fault-tolerant architecture for nuclear reactor and other critical process control applications*. The Twenty-First Annual International Symposium on Fault-Tolerant Computing, June 25-27, 1991, Montreal, Canada.

[2] D. Britxe, P. Traverse. *AIRBUS A320/A330/A340 Electrical Flight Controls. A Family of Fault-Tolerant Systems* The Twenty-Third International Symposium on Fault-Tolerant Computing, 22-24 Jun 1993.

[3] M. Merideth, A.K. Iyengar, T.A. Mikalsen, S. Tai, I.M. Rouvellou, P. Narasimhan. *Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications*. SRDS 2005 - 24th IEEE Symposium on Reliable Distributed Systems. IEEE, June 2005.

[4] K. Limaye, B. Leangsuksun, Z. Greenwood, S.L. Scott, C. Engelmann, R. Libby, K. Chanchio. *Job-Site Level Fault Tolerance for Cluster and Grid environments*. IEEE International Conference on Cluster Computing (Cluster 2005) Boston, Massachusetts, USA, September 27 - 30, 2005.

[5] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, R. Grima. *Programming Grid Applications with GRID Superscalar*. Journal of Grid Computing, 1(2):151-170, 2003.

[6] R. Raman, M. Livny, M. Solomon. *Matchmaking: Distributed Resource Management for High Throughput Computing*. Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing, July 28-31, 1998, Chicago, IL.

[7] I. Foster, C. Kesselman. *Globus: A Metacomputing Infrastructure Toolkit*. Int. Journal of Supercomputer Applications, 11(2):115-128, 1997.

[8] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, S. Matsuoka. *Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing*. Journal of Grid Computing, 1(1):41-51, 2003.

[9] *The home of the Checkpointing Packages*. http://checkpointing.org.

[10] S. Ghemawat, H. Gobioff, S. Leung. *The Google File System*. Proceedings of the nineteenth ACM symposium on Operating systems principles. pp 29 - 43. 2003. Bolton Landing, NY, USA.