Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Spalto Marengo 33, 15100 Alessandria
http://www.di.unipmn.it

TECHNICAL REPORT TR-INF-2003-12-07-UNIPMN
(October 2003)

Fault Tolerance in Grid Environment

Author: Massimo Canonico (canonico@mfn.unipmn.it)

Abstract

Computational Grids have the potential to become the main execution platform for high performance and distributed applications. However, such systems are extremely complex and prone to failures.

The scientific literature proposes different approaches to realize fault tolerance in Grid environments. In this paper, we present the state of the art describing the main projects studied.

1 Introduction

The Grid environment refers to the Internet-connected computing environment in which computing and data resources are geographically dispersed in different administrative domains with different policies for security and resource uses. The computing resources are highly heterogeneous, ranging from single PCs and workstations, cluster of workstations, to supercomputers. With Grid technologies it is possible to construct large-scale applications over the Grid environment.
Failures or error conditions due to the inherently unreliable nature of the Grid environment include hardware failures (e.g., host crash, network partition, etc), software errors (e.g., memory leak, numerical exception, etc) and other source failures (e.g., machine rebooted by the owner, network congestion, excessive CPU load, etc).
Grid applications should be able to handle failures which are sensitive to task context, what we call *task-specific failures*. Each task has its own failure semantics; that is, failure definition and failure handling strategies are specific to the task. These task-specific failures as well as failures in the Grid environment should be able to be detected and handled in a variety of ways depending on the execution semantics of both the task and overall Grid application:

- if the task is not completed within 30 minutes, allocate a new resource and restart;
- if not enough disk space remaining, terminate the task in advance, and either restart it on a machine with significantly more disk space, or retry in the same machine, but with a different algorithm which requires less disk space;
- for long running task, checkpoint periodically and, in case of failure, restart from the last good state;
- in case of a task running on unreliable execution environments, have multiple replicas of the task run on different machines, so that as long as not all replicated task fail, the task will succeed to execute;
- in other case, undo the effect of the failed task and retry.

The development of Grid application is difficult due to the complexity of both the underlying Grid environment and the application themselves. Coding manually those task-specific failure detection and failure handling procedures within the application is not a viable solution because it makes the design and development of Grid applications much more complicated. In addition, this approach requires application programmers to start from scratch by embedding fault tolerance procedures inside the application code in an *ad hoc* manner each time-consuming, error-prone, and inflexible. The scientific literature proposes different approaches to realize fault tolerance in Grid environments. In the next section, we present the state of the art describing the main projects studied.

2 Approaches

The main contributions about fault tolerance in Grid computing are Grid Workflow [1] (from University of Southern California) and WP4 European DataGrid project [2]. Let's describe them.

2.1 Grid Workflow System (Grid-WFS)

Grid Workflow is a flexible failure handling framework which addresses the requirements for fault tolerance in the Grid such as support for diverse failure handling strategies, separation of failure handling strategies from application codes, and task-specific failure handling. The Grid-WFS is implemented as a standalone application on top of the Globus Toolkit [6] v2.0. Grid-WFS consists of three major components:

- a Workflow Process Definition Languages using XML (XML WPDL) that allows users to define workflow process specification in a Directed Acyclic Graph (DAG) form;
- a Workflow Engine that controls workflow execution by navigating the workflow specifications, submitting tasks to specified Grid nodes, and monitoring the status of submitted tasks;
- Workflow runtime service that provide directory services necessary for the workflow engine to perform resource brokering during the workflow execution, including software, data and resource category services.

The Grid-WFS allows users to achieve failure recovery in a variety of ways based on the requirements or constrains of their applications following detection of the two failure levels (Figure 1):

- *Task-Level* techniques refer to recovery techniques that are to be applied in the task level to mask the effect of *task crash* failures. These techniques realize the so-called *masking fault tolerance* techniques such as *retrying, checkpointing* and *replication.*

- *Workflow-level* techniques refer to recovery techniques that enable the specification of failure recovery procedures as part of application structure. These techniques realize the so-called *nonmasking fault tolerance* techniques such as *alternative task*; basically, these techniques allow alternative task to be launched to deal with not only *user-defined exceptions* but also the failure that *task-level* techniques fail to mask (e.g., due to not enough redundant resources) in the task level.

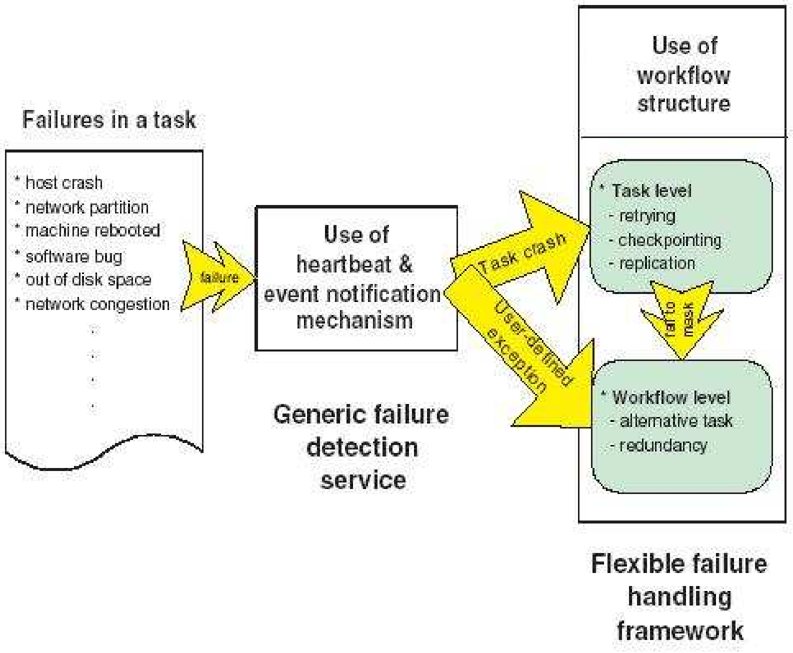


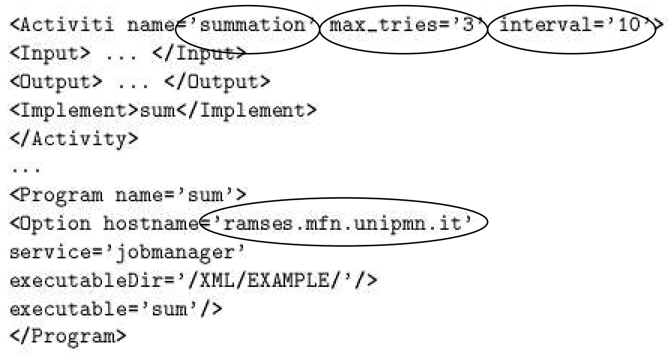
Figure 1: Overview of Grid-WFS

Some examples about how to realize these failure handling levels following.

Task-Level Failure Handling Examples

There are different way to prevent *task crash* failures:

Retrying: this might be the simplest failure recovery techniques to use in hope that whatever cause of failures will not be encountered in subsequent retries. Figure 2 illustrates a XML WPDL example of retrying.



```
<Activiti name='summation' max_tries='3' interval='10'>
<Input> ... </Input>
<Output> ... </Output>
<Implement>sum</Implement>
</Activity>
...
<Program name='sum'>
<Option hostname='ramses.mfn.unipmn.it'
service='jobmanager'
executableDir='/XML/EXAMPLE/'/>
executable='sum'/>
</Program>
```

Figure 2: This XML WPDL fragment describes that if the task crash failure is detected, this particular task named 'summation' would be retried on the specified Grid resource (i.e., whose hostname is "ramses.mfn.unipmn.it") up to 3 times with a interval of 10 seconds between tries.

Replication: the basic idea of this failure handling technique is to have replicas of a task run on different Grid resources, so that as long as not all replicated tasks crash (due to host crash, host partition away from the Grid client, etc), the task execution would succeed. The following XML WPDL example shows how specify 3 different resources where execute the 'summation' task:

```
<Activiti name='summation' policy='replica'>
 ...
<Implement>sum</Implement>
</Activity>
...
<Program name='sum'>
<Option hostname='ramses.mfn.unipmn.it'/>
<Option hostname='shrek.mfn.unipmn.it'/>
<Option hostname='alice.mfn.unipmn.it'/>
</Program>
```

Checkpointing: many checkpoint libraries and program development libraries which support checkpointing are available in Grid-WFS (i.e., Dome [3], Fail-safe [4] and CoCheck [5] are examples of such program). With these checkpointing facilities, checkpoint-enable applications can be developed simply by linking to them.

WorkFlow-Level Failure Handling Examples

The failure handling techniques that could be applied in the workflow level are:

Alternative task: a key idea behind this failure technique is that when a task has failed, an alternative task is to be performed to continue the execution, as opposed to the retrying technique where the same task is to be repeated over and over again which may never succeed. Also in this case there are specific XML tags that permit to define behaviors like *if the Fast_Unreliable_Task has failed, run the alternative task version that could called Slow_Reliable_Task*;

redundancy: as opposed to the task-level replication technique where same tasks are replicated, having multiple different tasks run in parallel for a certain computation is the basic idea of this technique. Thus, as long as at least one task has finished successfully, then the computation would succeed;

user-defined: this technique allows users to give a special treatment to a specific failure of a particular task. For example, the *Slow_Realiable_Task* is specified to be activated to handle a task-specific failure (i.e., a user-define exception called *disk_full*) that might arise during the execution of *Fast_Unreliable_Task*. Some experiments show that in heterogeneous computing environments like the Grid, it appears to be essential to support multiple fault tolerance techniques and user-defined exception handling in order to achieve high performance as well as fault tolerance in presence of failures.

Grid-WFS demonstrates trough experiments [1] that in heterogeneous computing environments like the Grid, it appears to be essential to support multiple fault tolerance techniques and users-defined exception handling in order to achieve high performance as well as fault tolerance in the presence of failures.

2.2 DataGrid: Fabric Monitoring and Fault Tolerance

The DataGrid Project aims at enabling next generation scientific exploration which requires intensive computation and analysis of shared large-scale databases(millions of Gigabytes), across widely distributed scientific communities. "DataGrid" is funded by European Union. In this project, the *Work Package 4(WP4)* is dedicated to the *Fabric Monitoring and Fault Tolerance* (FMFT) that provides the framework for monitoring of performance, functional and environmental changes for all resources contained in a fabric. The framework contains a global distributed repository for all monitoring measurements and a well-defined interface to plug-in data analysis routines (Correlation Engines) that regularly check that measurements are within configured limits and trigger alarms or automatic recovery actions in case they are not. The FMFT system consists of two parts:

- the monitoring framework for gathering, transporting, storing and accessing monitoring information;
- a basic set of monitoring sensors, fault tolerance correlation engines and the recovery actuators.

The advantage of using monitoring is to get a history record of changes and their timestamps, which will allow for detailed tracing of problems. The monitoring measurements are stored in such a way as to allow for efficient retrieval with a triplet key *(node, metrics, time)*. The impact of running the monitoring and fault tolerance components on the monitored resources must be under control of the monitoring system itself, limiting the resource utilization of the controlled sensors.

Functionality

The monitoring framework part of the FMFT subsystem consists of:

Monitoring Sensor Agent (MSA): the MSA is responsible for calling the monitoring sensors, receiving the measurement data from the sensors and assuring that the data is forwarded to the Measurement Repository;

A Measurement Repository (MR): the MSAs insert the monitoring measurements into the MR where the information is stored together with a timestamp. The MR consists of a client API and a global repository server. The client API provides methods for inserting data in the repository and a metric-subscription/notification mechanism for clients to subscribe to metrics and be notified every time those metrics have been measured. The latest measurements are cached in persistent local storage (disk), which ensures autonomy for the Fault Tolerance components that are local to the node in case the network is unreachable.

Monitoring User Interface (MUI): the MUI provides an easy-to-use graphical interface to the measurement repository. It automatically queries the Configuration Management subsystem for high-level configuration information and presents the user with comprehensive views of the monitoring information, for instance health and status displays for entire services rather than individual nodes.

Monitoring Sensor (MS): An MS is an implementation of the MonitoringSensor interface that performs the measurements of one or several metrics. The MS is typically driven by rules stored in the Configuration Management subsystem. A given MS implementation can thus be used for several similar metrics, e.g. a single daemon dead sensor can be used to monitor the running of several daemons. The MS provides the plug-in layer for any data producer to insert its data into the monitoring system.

The fault tolerance part of the FMFT subsystem consists of:

Fault Tolerance Correlation engine (FTCE): The FTCE is the active correlation engine of the FMFT subsystem. The FTCE runs as a daemon process on all nodes and should be implemented to be robust to most system component failures. The FTCE processes measurements of one or several metrics stored in the MR to determine if something has gone wrong or is on its way to go wrong on the the system and if so, determines what recovery actions are needed, and call the Actuator Dispatcher to launch the those actions. The FTCE implements the MonitoringSensor interface and is sampled by the MSA as a normal sensor. The output metrics values contain normally a Boolean that reflects if any fault tolerance actuators were launched, and if so, the identifiers of the actuators and their return status. The FTCE processing for a given metric is triggered either through a periodic sampling request from the MSA or through the metric - subscription/notification mechanism provided by the MR;

Actuator Dispatcher(AD): The AD is used by the FTCE to dispatch fault tolerance actuators. It consists of a client API and an agent that controls all actuators on a local system. The AD agent does not maintain any permanent channel to the requester. Instead the client API returns a unique handle for each dispatch request and is able to return the status of any dispatched actuator given the unique handle. The completion status of a dispatched actuator can be retrieved asynchronously. The running of the actuators is serialized so that only at most one actuator can run at any given time. The received requests are queued for FIFO scheduling. Certain dispatcher requests, e.g. immediate shutdown due to hardware failure, are allowed to bypass the normal queue;

Fault Tolerance Actuator (FTA): An FTA is an implementation of the FaultToleranceActuator interface that executes automatic recovery actions. The FTA is typically driven by rules stored in the Configuration Management subsystem. A given FTA implementation can thus be used for several similar recovery actions, e.g. a single daemon restart FTA can be used to call a restart method in the Software Package (SP) class of all software packages that are installed on the node. The

FTA is dispatched by the AD agent. Since the FTA may trigger a reboot of the system, there is no open channel between the FTA and the AD agent and the return status must be stored in permanent local storage. For the same reasons the FTA notifies the AD agent when the return status is available.

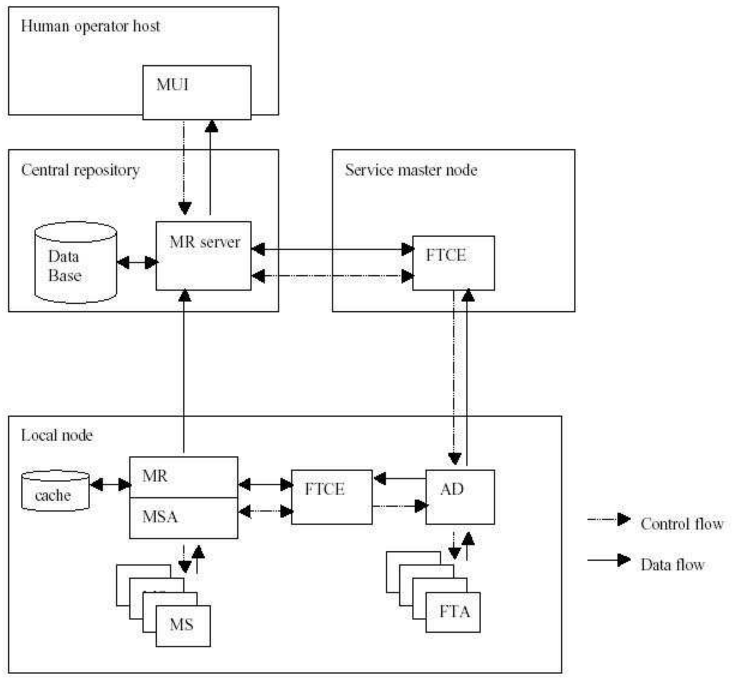


Figure 3: Deployment view of the fabric monitoring and fault tolerance components

Figure 3 shows the deployment view of the fabric monitoring and fault tolerance components together with the information flow. As can be seen the MR part managing the cache on the local node is contained in the same process as the MSA. The FTCE runs both on locally and centrally. The local FTCE handles all actions, which can be decided locally. The central FTCE runs on, for instance, a cluster master, and it handles actions, which have to be correlated between several nodes (e.g. all CPU nodes in the cluster).

3 Conclusion

Fault tolerance is one of the most important open problem in Grid environment but the scientific literature proposes only two projects.

The system presented in the DataGrid proposal is dated November 2001. After this date no more documentation has been produced and there isn't a implementation of what they proposed. Besides, the documentation is really superficial:

- it does not explain what measurements are stored in the Measurement Repository(MR);
- it does not explain how FTA (Fault Tolerance Actuator) detects a fault querying the MR;

On the contrary, the Grid-Workflow project is definitely in an advanced status. The system describes in section 2.1 has been implemented as standalone application over Globus Toolkit v.2.0 (GTK2.0). Besides, the researchers of University of Southern California have tested the Grid-Workflow in a simulated environment [1]. The simulations have provided interesting results, varying different parameters (i.e., fault rate, checkpoint time, replications number) using different application (i.e., jobs with long/short completion time). The results show that:

- for high failure rate the best fault tolerance technique is checkpointing with replication;
- for low failure rate the best fault tolerance technique is only replication;

In the second case, the use of checkpointing with or without replication mechanism as a recovery policy appears to be inappropriate due to the checkpoint overhead. Different tasks needs different techniques to manage faults.
A contribution for the fault tolerance in Grid environment could be the study and the development of an infrastructure that provides Grid applications with QoS requirements concerning not only absolute performance, but also reliability/availability requirements. More specifically, this infrastructure will include a set of fault tolerance mechanisms able to provide the desired reliability/availability level, a monitoring component in charge of determining the overhead that the various mechanisms induce on application execution, and a scheduler able to select, for a given application and QoS requirements, the set of resources that best satisfy the application needs. Traditionally performances and reliability are considered two independents aspects, whose interaction is not considered when choosing the resources where the application will run. Consequently, if the chosen resources become unavailable after the execution is started, application performance may be much lower than expected. To solve this problem, we will pursue a different approach in which reliability and performances are simultaneously taken into account when making scheduling decisions. More specifically, we will develop a software infrastructure supporting different fault tolerance mechanisms (based on hardware and/or software replication), that will include a

scheduler able to choose automatically, for each application, the best suited mechanism to its needs. In our approach the user will be given the possibility of specifying the quality of service both in terms of performance and reliability (for instance, (s)he could require that the application completes its execution within a given time with a given probability). The scheduler will use the information concerning the real status of the resources, their expected availability, and the overhead of each fault tolerance mechanism to select the set of resources that satisfy to the maximum possible extent the user QoS requirements. If these requirements cannot be met, the scheduler will compute some alternatives, that may be inferior in terms of performance, reliability, or both, that will be presented to the user, that in this way will be able to decide whether it is better to accept this lower QoS and start the application immediately, or to wait until additional/different resources become available. Besides, each resource has a utilization cost and each user has a available budget. The scheduler must choose the "best resource" for each job considering performances, reliability but also the available budged of the job submitter. To develop the scheduler, we plan to investigate the use of mathematical models, based on formalisms of various nature (e.g., Petri nets and queuing networks), whose parameters will be instantiated by measurements collected by means of a suitable monitoring infrastructure.

References

[1] S. Hwang, C. Kesselman. *Grid Workflow: A Flexible Failure Handling Framework for the Grid.* June 2003.

[2] G. Cancio, O. Barring. *DataGrid: Architectural design and evaluation criteria. WP4 Fabric Management.* November 2002.

[3] A. Beguelin, E. Seligman, and P.Stephan. *Application level fault tolerance in heterogeneous networks of workstations.* June 1977.

[4] J. Leon, A. L. Fisher, and P. Steenkiste. *Fail-safe pvm: A portable package for distributed programming with transparent recovery.* February 1993.

[5] G. Stellner. *Cocheck: Checkpointing and process migration for mpi.* April 1996.

[6] *http://www.globus.org.*