

A SERVICE FOR RELIABLE EXECUTION OF GRID APPLICATIONS

Elżbieta Krępska and Thilo Kielmann

Dept. of Computer Science

Vrije Universiteit

Amsterdam, The Netherlands

e.krepska@gmail.com

kielmann@cs.vu.nl*

Raül Sirvent and Rosa M. Badia

Barcelona Supercomputing Center and

Universitat Politècnica de Catalunya

Barcelona, Spain

rsirvent@ac.upc.edu

rosab@ac.upc.edu

Abstract In grid environments, with the large number of components (both hardware and software) that are involved in application execution, the overall probability that at least one of these components is (temporarily) non-functional is increasing rapidly. In traditional operating systems, such failures are flagged as fatal and the application will be stopped, relying on a re-start after the problem will have been fixed. In a large grid system, this is not a feasible approach as failures happen too frequently while error diagnostics might not be possible at all.

This scenario is asking for a different approach to application execution, where detection and circumvention of error conditions become an integral part. We present a service that is keeping track of an application's life cycle, from submission by the user to successful completion of its execution. In a case study, we describe how GRID superscalar, a grid application programming environment, can benefit from our service.

Keywords: Reliable execution service, fault tolerance, mediator component toolkit, GRID superscalar.

*Contact author

1. Introduction

Grid environments integrate diverse resources and services. Also, the number of nodes in a grid might be highly variable as, for example, in the LHC Computing Grid (LCG) [26]. Such heterogeneity and churn contribute to the complexity of a grid environment, leading to some degree of unreliability. Geographic dispersion, along with administrative site autonomy [10] further contribute to complexity and thus unreliability. Performance variability and sporadic unavailability of the underlying networks complicate matters even further.

Obviously, with a large number of components (both hardware and software) involved in the execution of a grid application, the overall probability that at least one component is (temporarily) non-functional is increasing rapidly. In traditional systems, such failures are flagged as fatal and the application will be stopped, relying on a re-start after the problem will have been fixed.

In a large grid system, this is not a feasible approach as failures happen too frequently while error diagnostics might not be possible at all, either because of the overall system scale, or because logging information might be available only locally at the system that has failed.

This scenario is asking for a different approach to application execution, where detection and circumvention of error conditions become an integral part. In previous work [17], we have identified the need for a persistently running service that is keeping track of an application's life cycle, from submission by the user to successful completion of its execution. In this paper, we present such a service.

In Section 2, we investigate failure types and fault-tolerance mechanisms. Design and implementation of our reliable execution service (RES) will be presented in Section 3. Section 4 describes how RES can be used for executing application workflows from the GRID superscalar system. Section 5 concludes.

2. Failures in Grid Environments

Typically, a user submits his application via a resource broker to a machine in the grid. Before the job is started, required input files must be pre-staged (copied to the execution host) and afterwards, output files have to be post-staged to the user. Failures might happen during many stages of that process:

- 1 *Execution services failure.* Services executing the application such as a resource broker or a local scheduler might misbehave/crash or might not be able to satisfy the job requirements.
- 2 *Local environment failure.* The application might fail locally on the selected machine, for example because of insufficient disk space.
- 3 *Resource failure.* A resource might fail that the application depends upon, like network outages or unresponsive services.

- 4 *Application specific failure.* The application might fail due to a problem with its code, like causing memory segmentation faults, numerical errors or deadlocks.

Failure (1) of a resource broker or a scheduler might be masked by trying to start a job on a different site. A failure of grid job manager can be circumvented by submitting directly to a cluster, provided we can discover hosts belonging to a grid. Failure (2) can be masked by restarting a job on a different host. To discover failure type (3), special failure detection and monitoring services are necessary [6, 9, 13, 16, 21]. Only failures of type (4) should not be masked, and feedback about their occurrence should be provided to the user. Fault-tolerance (FT) mechanisms can be implemented on different levels:

Application-level FT. Here, fault tolerance is built directly into the application code. While this approach can provide flexibility and efficiency by exploiting application-level knowledge, it is often perceived as cumbersome by application programmers [19]. Runtime environments like the GAT [1] or Satin [24], or fault-tolerant versions of MPI [8] can lower the burden on the programmer.

Multi-layered FT. Here, each layer handles the errors within its own scope, pass others up the hierarchy [23]. This approach simplifies higher-level layers, however, at the expense of performance penalties as higher layers themselves can not adapt any more [18]. Building FT into each component in a grid is also not practical, due to scale and diversity of components.

External FT services. The grid environment might offer an automatic failure detection service which basically allows building more intelligent fault-tolerant services on top of them. Such services [6, 21], however, suffer from poor integration with application execution and from the necessity to be ubiquitously deployed in a grid.

2.1 Related Work

Phoenix is detecting failures by scanning scheduler log files [18]. It can diagnose execution and data transfer errors. Phoenix can follow different, user-definable failure-handling strategies. Applicability of Phoenix is limited to those systems of which log files can be interpreted.

Application failures can also be handled on workflow level [14]. Here, individual tasks might be run alternatively should another task fail. The system in [14] relies on information from the resource broker only, but can also be combined with heartbeat monitors [13]. The execution service for the NASA Information Power Grid [20] is executing interdependent tasks, restricting error diagnosis to application exit codes.

The Globus GRAM service [25] is frequently criticized for not returning application exit codes, returning its own codes specifying certain types of fail-

ures instead. Condor-G [11] uses the GRAM protocol and detects and handles resource failures to provide "exactly once" execution semantics. Unfortunately, this mechanism can not be used without deploying Condor-G as the grid middleware.

The fault tolerant manager proposed in [3] supervises the execution of jobs and in case of a failure a decision maker decides on a recovery scenario. Interestingly, the proposal counts a dramatical decrease in the application performance as a failure, too.

Some systems focus on failure detection only. The Heartbeat Monitor (HBM) [21], is an unreliable, low-level fault detection service for a distributed system. The service bases on unreliable fault detectors [4] which try to discover which system components (machines or processes) *might* have failed and notify the application of that fact. Whether to trust this information, how to interpret it, what to do about it, is left entirely to the application. Défago et al. [6] improves performance and scalability of this scheme by introducing hierarchisation and gossiping between monitors.

However, those services are lower-level, they only inform that they suspects that a component has crashed. The application should decide whether and how to recover from the failure what can be non-trivial, especially that the heartbeat monitors do not know the failure root cause. In the following, we describe a service that implements a rather comprehensive approach to failure detection, diagnosis, and resolution.

3. A Service for Reliable Application Execution

In previous work, we have identified the need for a persistently running service that is keeping track of an application's life cycle, from submission by the user to successful completion of its execution [17]. Such a service is supposed to become an integral part of a grid application execution environment based on a mediator component toolkit [5]. We will now present design and implementation of RES, our *Reliable Execution Service*, that has been designed to fulfill this purpose.

3.1 Design objectives

The Reliable Execution Service (RES) is a permanent service providing reliable execution of applications submitted to the grid. Permanence means that the service runs all the time. Reliability means handling transient failures transparently to the end user. The user should observe only the very application errors, with the exception of permanent grid environment failures. The RES service will be designed to meet the following requirements:

Application feedback. In an ideal case, there should not be any requirements posed on the executed application. However, having application feedback enhances greatly failure detection capabilities and enables application specific debugging and logging. Therefore, for both service-aware and unaware applications, the reliable execution functionality should be offered. Additionally, an application can talk to the service, for example requesting an application-specific failure or sending a message to the user.

Failure detection. The service should detect submission and execution problems. After the execution is done it should properly detect the exit code and intercept uncaught exceptions whenever possible. If a monitoring service is available, the service might monitor hosts executing the application, the application process or resources that the application depends on.

Failure handling and recovery. The service should offer diverse failure handling strategies. Checkpointing applications must be supported.

User influence. The user might influence failure detection and recovery schemes by specifying policies. Policies should be separated from the application code.

Workflow-level and task-level fault-tolerance techniques. The service will not implement task-level fault tolerance techniques, such as alternative or redundant tasks (see Section 2.1). However, the service's direct API should allow a developer to implement those techniques easily.

Batch-mode application execution. After submitting a job, the user can go offline and access recorded job status information at a later time. Job history is kept during and after job completion. The user should not have to examine output files to check the application status or to see if it exited correctly.

Minimizing administrative overhead. The service should be designed so that special privileges, accounts or certificates are not necessary. The service should use the deployed security framework and require users to delegate their rights for the service to act on their behalf. Also, there should be no need to install anything on each machine in the grid.

Preventing information loss. Masked grid component failures should be transparent to the user but not to grid middleware or to the administrator. Records of failure occurrences should be stored persistently, to allow other components to adapt their behaviour.

Dependability. The service should be able to sustain a crash of its hosting machine. When the machine is back online, the service should recover and re-acquire jobs activated before the crash.

Portability. The service should be independent of the specific middleware installed on the grid, the hosting machine platform, the language-binding of an executed application and the environment of targeted machines. The service acts as a grid meta-scheduler (is able to choose a cluster to run a job), so it has to encapsulate specific grid information at least on clusters belonging to the grid. The encapsulation should be clearly separated from the service code and it should be possible to discover this information dynamically.

Job model. The service is intended for computation intensive, long running applications (long here means for example more than an hour). The application file staging model is simple as well – it consists of two lists of files to pre- and post-stage. Additionally, for checkpointable applications, the application specification might contain a list of names of checkpoint files and a checkpoint files repository location.

3.2 Service design

The architecture of the Reliable Execution Service is presented in Fig. 1. The most important thread of control in the service is the **JobController**. It handles users' requests, activates jobs and controls them. It saves execution metadata to the **JobRepository**. Waiting jobs are submitted to the grid via the **GATEngine** [1], which chooses an appropriate resource brokerage service to submit the job to.

Job's **Executors** are threads of control active on grid machines. They supervise the execution "locally", give feedback to the service and mediate communication between the service and the application. After the job is done, its **Executor** detects the exit code and uncaught exceptions, sends a **JobDone** signal to the service and exits. The service verifies the execution. In case of a failure, the failure agent diagnoses the problem, classifies it into permanent or transient. Permanent failures are reported to the user in the way that shields the user from middleware and hardware details. For transient failures the agent comes up with a suitable recovery scenario executed then by the **JobController**.

In case of a failure of the machine hosting the service, the service itself might stop working. However, it is designed such that the previously activated jobs are reacquired, even though control over them is weakened, as connections with the resource broker are lost that were created when the jobs had been submitted.

Service behaviour such as the job execution planning phase, failure diagnosis and execution verification might be influenced by users policies, service

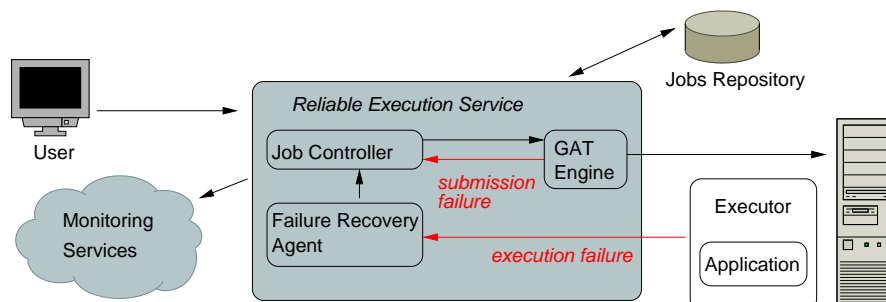


Figure 1. The Reliable Execution Service architecture.

self adaptation based on failures occurrence rates and additional information received from monitoring or information grid services.

As the service is executing on the behalf of its users, it has to use their respective credentials. Our current implementation is using the Globus MyProxy Credential Management System [12]. Using MyProxy, the user can hand over temporary credentials to the RES service, allowing RES to execute the application with the user's credentials. Other security contexts, supported by the GAT engine, could be also used here.

However, delegating POSIX (UNIX) file permissions could not be done easily. Therefore, the user must make sure that the service can access files which are to be pre-staged: the service might be run by a privileged user or the files might be made readable to the appropriate user group or to everybody. An appropriate solution would be accessing files through a grid file system abstraction, as anticipated in [15], with proper grid-wide file permissions.

3.3 Application wrapping

Applications are not executed directly but they are wrapped inside the Executor which is responsible for the following tasks:

- 1 *Controlling the application execution.* The wrapper signals to the service when the execution begins, runs the application and after it is done, reports this fact to the service.
- 2 *Local monitoring of the application execution.* During the execution the wrapper sends heartbeats to the service. When the job is done, it detects the exit code and intercepts uncaught exceptions if possible.
- 3 *Talking to the application.* The Executor mediates the communication between the application and the service. The application might send signals, which are interpreted by the JobController according to the user's policy. The signal might be fatal, i.e. describing a failure which

occurred during the execution, or informative, for example debugging and logging signals.

The information received from the `Executor` might be more accurate and more timely than information from the resource broker which does not control the application directly. Also this is the only way to correctly detect and propagate the exit code of the application.

Two kinds of `Executor`'s were implemented: a `GenericExecutor` that can execute any application, and a `JavaExecutor` that can execute only applications implemented in Java but provides more functionalities for them. Communication with the service is done using Java RMI. As executed applications might be developed in any programming language, it was not trivial to implement a generic way to talk to the application. As the `JavaExecutor` runs the application using the Java's reflection mechanisms, it talks to the application through normal local method invocations. However, the `GenericExecutor` talks to the application using sockets. To ease the application programmer's task, we have implemented simple language-binding libraries that a user can link to his programs, for in C/C++ and Python languages. If the application is only available as a binary program, it cannot be modified to talk to the `Executor`. In this case, the `Executor` can only handle return codes.

Fig. 2 shows some example code of an application that uses the RES programming interface to try an alternative algorithm should the first one fail. The code first tries a supposedly fast implementation. Should this fail, another (possibly slower but less resource hungry) implementation is tried out. This is an example where certain application-level failures can be circumvented using RES.

3.4 Failures detected by the service

The most important detected failures, related to the fact that the application executes in the grid environment, are as follows:

- *Failures prior to the execution.* Before the execution on the target machine takes place at all, numerous types of failures must be anticipated such as pre-staging failures, incorrect job description, security failures and multiple types of submission failures – inability to locate the service, service misconfiguration, not enough resources to execute the job, authorisation/proxy failure, network connection failure, etc.
- *Failures during the execution.* Besides application specific failures such as abrupt termination, throwing an uncaught exception or a failure requested by the application, failures might be caused by other factors such as the inability of the `Executor` to start the application, security issues, problems with talking to the application or the `Executor` internal problem

```

boolean submitAndWait(RESSoftwareDescription sd, RESUserPolicy policy) {
    // Locate the service
    RemoteRES res = RESServiceLocator.find();
    // Submit the job
    RESSubmissionResult sr = res.submit(sd, policy);
    // Retrieve the resulting job id
    RESJobId jobId = sr.getJobId();
    // Wait until job is done
    RESQueryResult qr = null;
    while (!(qr = res.query(jobId, true)).isDone());
    // Check if job is done successfully
    return qr.isDoneSuccessfully();
}

// Prepare software descriptions and user policies for both algorithms
// Try the first algorithm
if (!submitAndWait(resSD_fast, userPolicy_fast)) {
    // The first algorithm failed, try the second one
    submitAndWait(resSD_slow, userPolicy_slow);
}

```

Figure 2. Example code for submitting alternative tasks based on given errors.

which should not – but might – occur and in this case should not mislead the user. The service is capable of detecting an abrupt termination of the application basing on the heartbeats received from the **Executor**. When detecting missing heartbeats the service tries to differentiate between crash of the job process or the **Executor**, a crash of the machine executing the application, or a transient or permanent network outage.

- *Post execution failures.* After the job execution, the service anticipates post-staging failures and verifies the execution according to the following criteria: heartbeats reception, feedback signals reception, the exit value, presence of output files.

3.5 Recovery techniques

When the **FailureRecoveryAgent** detects a failure, it must come up with a suitable recovery scenario to resolve the problem. As we cannot prevent grid resources from failing, what we can basically do at the task level is restarting the application in the way that diminishes the probability of the failure recurring. Therefore we mask transient failures using various types of **RETRY** techniques: retrying on a different cluster (what provides hardware and software diversity), retrying on a machine with certain characteristics, such as a faster processor,

more disk space, more memory or with more network bandwidth available or restarting after a certain time.

Another technique used is CHECKPOINTING. It can be used only if the application is developed so that it can write checkpoints or the grid provides system-level generic checkpointing. The stored state information should be enough to restart the process, even on a different resource. This technique is useful for processes which involve significant data manipulation and are unstable or cannot finish data manipulation in a single run.

If the application fails a certain number of times, the failure must be considered permanent. The application FAILURE is reported back to the user. Also in certain cases Executor failures might be worthy to IGNORE, as we may still hope that the application executes successfully.

To select a proper recovery scenario, artificial intelligence techniques might be used, to base the decision also on the job history and overall failure occurrence rates. In the current implementation the history of the job is not taken into account when deciding on a recovery scenario. Misbehaving clusters are detected when failure occurrence rate is near 100% and jobs are not submitted to them provided there are other resources to use.

3.6 Evaluation

The service has been implemented according to the design shown in Fig. 1. We have evaluated its functionality with a fault-simulating application kernel that we had pre-set to crash with a given probability (between 10 and 20%). We used up to 100 worker tasks running the fault-simulating application kernel and up to 32 compute nodes. When a worker crashed, it was automatically restarted by the service, transparently to the application. All of the workers completed eventually in spite of the frequent crashes. In a more complete set of tests, we have verified that RES is able to handle the following kinds of failures:

User-related failures: authorisation failures, program binary unavailability, non-zero exit code, pre staging failures due to unavailable files or target directories, catching the application's exceptions, application timeouts, handling of checkpoints.

Grid-related failures: crash of the Executor process, transient network outage, host crash or permanent network outage, cluster/scheduler misbehavior, crash of the RES service itself.

In a separate test, we have evaluated the runtime overhead of submitting jobs via the RES, compared to direct job submission. Fig. 3 shows the results, obtained on the VU cluster of the DAS-2 [7] system. We were experimenting with an application kernel performing prime factorization, written in Java, and parallelized as a simple task farming program. The program was submitted via

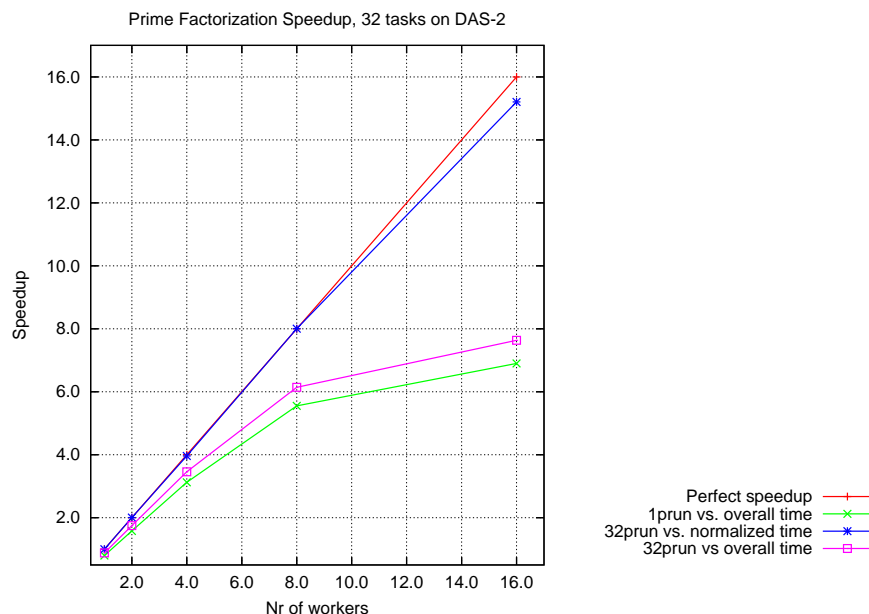


Figure 3. Job submission overhead caused by the RES.

the prun job submission tool installed on DAS-2. The RES contacted prun via the Java-GAT. In total, 32 parallel jobs were created per application run. The graphs in Fig. 3 report speedups, relative to sequential execution on a single node of the system.

The purpose of Fig. 3 is to verify the overhead imposed by the RES service, ranging from one up to sixteen worker nodes. In fact, the figure shows two comparisons. The first pair of lines to compare to each other are the ones labelled “1prun vs. overall time” and “32prun vs. overall time.” The lines compare submission of the 32 tasks, either as a single job, or by 32 separate job submissions, both to the RES. Notably, differences are only small. The second pair of lines for comparison are “32prun vs. normalized time” and the perfect speedup. Here, the runtimes have been normalized by removing the actual scheduling and waiting times of the underlying prun system. What can be seen is that speedups are (almost) perfect, indicating that RES introduces hardly any runtime overhead to the overall job submission process, while adding failure resilience.

4. Case Study: Reliable Execution for GRID superscalar

GRID superscalar [2] is a grid application programming environment, providing a simple programming model. Its runtime system is parallelizing se-

quential code, based on the functions defined to be run on the grid, and the data dependencies between the calls in the main program to those functions. The user provides a main program, an IDL file describing the interfaces of the functions which will be executed in the grid, and the function implementations. These can be either coded directly, or wrapped calling to external binaries or scripts. The system has been ported not only to grid environments (using Globus [25] and Ninf-G [22]), but also to work inside clusters with ssh/scp, and as a programming model for multi-core processor platforms, going beyond the boundaries of grid computing.

Recently, GRID superscalar has incorporated a fault-tolerance mechanism for which the user specifies timeout values for individual functions. When a timeout is reached, the runtime system assumes a function to have failed, and tries to cancel and resubmit it. In combination with the RES, GRID superscalar would only need to consider the application-related failures. Also, GRID superscalar can define suitable policies for specific failure scenarios, and, because the RES relies on the GAT framework, it indirectly offers a standard interface for using different grid middlewares transparently. In general, the RES handles the vast majority of grid middleware related errors, and thus simplifies GRID superscalar's task. In addition, the RES provides a uniform interface for worker tasks to report their exit status to the master process, obsoleting the current file-based mechanism that had been introduced to circumvent limitations of current middleware.

GRID superscalar's code generation and deployment techniques perfectly fit the RES framework. Before running an application, different binaries are generated, one for the master part, and several for the worker machines. To the RES, these binaries are like independent applications which must be invoked, along with their input and output files, while GRID superscalar keeps track about all of them, because they belong to the same application.

The checkpointing capabilities of the RES system are focused to offer an intra-task checkpoint mechanism. This is very suitable to the GRID superscalar runtime, which already offers checkpointing techniques, but based on an inter-task approach. So, the combination of the checkpointing inside a task offered by RES, and the checkpointing between tasks offered by GRID superscalar will build a robust framework for the execution of GRID superscalar's applications.

When submitting a job, GRID superscalar maintains information about file locality for input files, aiming to reduce file transfer overhead. This implies that part of the job submission policies (provided by GRID superscalar to RES) have to give preference for execution hosts with good locality of input files.

5. Conclusions

The large number of components being involved in the execution of a grid application raises the probability that at least one of these components is (temporarily) non-functional up to a level where errors can not be neglected any more. This observation calls for an application execution mechanism that has failure detection, diagnosis, and handling as intrinsic functionalities.

We have explored the possible causes of application execution errors, and existing approaches to handle them. We have presented the design and implementation of RES, our *Reliable Execution Service* for grid applications. We have presented a case study in which the GRID superscalar system reliably executes jobs via the RES service, and how the service also helps GRID superscalar to build its own fault tolerance policies. The combination of checkpointing techniques available in both environments creates a strong reliable framework where the computation already performed will be hardly lost.

RES is experimentally being deployed on the DAS-2 system. While we are currently gaining further experience with the system, future work comprises support for parallel jobs as well as more sophisticated failure resolution strategies, possibly based on AI techniques.

Acknowledgments

This research work is carried out under the FP6 Network of Excellence *CoreGRID* funded by the European Commission (Contract IST-2002-004265). It is also partially funded by the Ministry of Science and Technology of Spain (TIN2004-07739-C02-01).

References

- [1] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schuett, E. Seidel, and B. Ullmer. The grid application toolkit: toward generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, 93:534–550, 2005.
- [2] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.
- [3] M. Bubak, T. Szepieniec, and M. Radecki. A proposal of application failure detection and recovery in the Grid, 2003. Cracow, Grid Workshop.
- [4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
- [5] CoreGRID Institute on Problem Solving Environments Tools and Grid Systems. Proposal for Mediator Component Toolkit. CoreGRID deliverable D.ETS.02, 2005.
- [6] X. Défago, N. Hayashibara, and T. Katayama. On the design of a failure detection service for large-scale distributed systems. *Proceedings International Symposium Towards Peta-Bit Ultra-Networks*, pages 88–95, 2003.

- [7] The Distributed ASCI Supercomputer DAS-2. <http://www.cs.vu.nl/das2>.
- [8] G. E. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. *Lecture Notes in Computer Science*, 1908:346–354, 2000.
- [9] P. Felber, X. Défago, R. Guerraoui, and P. Oser. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, pages 132–141, Edinburgh, Scotland, 1999.
- [10] I. Foster. What is the Grid? A three point checklist. *GRID Today*, 2002.
- [11] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, 2001.
- [12] Globus Toolkit. The MyProxy Credential Management Service. <http://grid.ncsa.uiuc.edu/myproxy>.
- [13] S. Hwang and C. Kesselman. A generic failure detection service for the Grid. Information Sciences Institute, University of Southern California. Technical Report ISI-TR-568, 2003.
- [14] S. Hwang and C. Kesselman. Grid workflow: A flexible failure handling framework for the Grid. *High Performance Distributed Computing*, 00:126, 2003.
- [15] A. Jagatheesan. The GGF Grid File System Architecture Workbook. Grid Forum Document, GFD.61, 2006. Global Grid Forum.
- [16] A. Jain and R. K. Shyamasundar. Failure detection and membership management in grid environments. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 44–52, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
- [17] T. Kielmann, G. Wrzesinska, N. Curre-Linde, and M. Resch. Redesigning the SEGL Problem Solving Environment: A Case Study of Using Mediator Components. In *Integrated Research in Grid Computing*. Springer Verlag, 2006.
- [18] G. Kola, T. Kosar, and M. Livny. Phoenix: Making data-intensive grid applications fault-tolerant. In *5th IEEE/ACM International Workshop on Grid Computing*, 2004.
- [19] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauve. Faults in grids: Why are they so bad and what can be done about it? In *Fourth International Workshop on Grid Computing*, page 18, Los Alamitos, CA, USA, 2003. IEEE Computer Society.
- [20] W. Smith and C. Hu. An execution service for grid computing. NAS Technical Report NAS-04-004, 2004.
- [21] P. Stelling, C. DeMatteis, I. T. Foster, C. Kesselman, C. A. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.
- [22] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.
- [23] D. Thain and M. Livny. Error scope on a computational grid: Theory and practice. In *11th IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, CA, USA, 2002. IEEE Computer Society.
- [24] G. Wrzesinska, R. van Nieuwpoort, J. Maassen, and H. E. Bal. Fault-tolerance, malleability and migration for divide-and-conquer applications on the grid. In *Proc. of 19th International Parallel and Distributed Processing Symposium*, Denver, CO, April 2005.
- [25] The Globus project. <http://www.globus.org>.
- [26] LHC Computing Grid (LCG) project. <http://lcg.web.cern.ch/LCG>.