Applications Resilience on Clouds

Toan Nguyen, Jean-Antoine Desideri, Laurentiu Trifan

Applications Resilience on Clouds

Toàn Nguyên, Jean-Antoine Désidéri, Laurentiu Trifan
Project OPALE
INRIA
38334 Saint-Ismier, France
Toan.Nguyen@inria.fr

Abstract— **Cloud computing infrastructures support system and network fault-tolerance. They transparently repair and prevent communication and software errors. They also allow duplication and migration of jobs and data to prevent hardware failures. However, only limited work has been done so far on application resilience, i.e., the ability to resume normal execution after errors and abnormal executions in distributed environments and clouds. This paper addresses open issues and solutions for application errors detection and management. It also overviews a testbed used to to design, deploy, execute, monitor, restart and resume distributed applications on cloud infrastructures in cases of failures.**

Keywords- Resilience; Cloud Computing; Scientific Applications; High-Performance Computing; Workflows.

I. INTRODUCTION

This paper overviews open issues and the corresponding solutions for application errors detection and management when running on cloud infrastructures. A testbed is also presented relying on a workflow system interfaced with a grid infrastructure to model cloud environments. Section II addresses error detection. Section III details the corresponding solutions for error management. Section IV describes the testbed using a workflow management system to support these solutions on clouds. Section V is a conclusion.

Because many terms are used in the fault-tolerance area, a definition of various terms used in the domain is necessary. An interesting definition of errors, faults and failures is given in a system such as Apache's ODE [11], where system *failures* and application *faults* address different types of errors.

A. Errors

The generic term *error* is used to characterize abnormal behavior, originating from hardware, operating systems and applications that do not follow prescribed protocols and algorithms. Errors can be fatal, transient and warnings, depending on their criticity level. Because sophisticated hardware and software stacks are operating on all production systems, there is a need to classify the corresponding concepts (Figure 1).

B. Failures

A failure to resolve a DNS address is different from a process fault, e.g., a bad expression. Indeed, a system failure does not impact the correct logics of the application process at work, and should not be handled by it, but by the system error-handling software instead: "*failures are non-terminal error conditions that do not affect the normal flow of the process*" [11].

C. Faults

However, an activity can be programmed to throw a fault following a system failure, and the user can choose in such a case to implement a specific application behavior, e.g., a number of activity retries or its termination.

Application and system software usually raise *exceptions* when faults and failures occur. The exception handling software then handles the faults and failures. This is the case for the YAWL workflow management system [19][20], where specific *exlets* can be defined by the users [21]. They are components dedicated to the management of abnormal application or system behavior (Figure 2). The extensive use of these exlets allows the users to modify the behavior of the applications in real-time, without stopping the running processes. Further, the new behavior is stored as a component workflow which incrementally modifies the application specifications. The latter can therefore be modified dynamically to handle changes in the user requirements.

D. Fault Tolerance

Fault-tolerance is a generic term that has long been used to name the ability of systems and applications to handle errors. Transactional systems for example need to be fault-tolerant [9]. Critical business and scientific applications need to be fault-tolerant, i.e., to resume consistently in case of internal or external errors.

E. Checkpoints

Therefore *checkpoints* need to be designed at specific intervals to backtrack the applications to consistent points in the application execution, and *restart* be enabled from there. They form the basis for *recovery* procedures.

In the following, we call *checkpoint* for a particular task the set including task definition, parameter specifications and

data associated to the task, either input data or output data and the parameter values.

This checkpoint definition does not include the tasks execution states or contexts, e.g., internal loop counters, current array indices, etc. Therefore, we assume that checkpointed tasks are stored stateless. This means that interrupted tasks, whatever the reasons and errors, cannot be restarted from their exact execution state immediately prior to the errors.

F. Recovery

We assume therefore that the recovery procedure must restart the failed tasks from previously stored elements in the set of existing tasks checkpoints. A consequence is that failed tasks cannot be restarted on the fly, following for example a transient non fatal error. They must be restarted from previously stored checkpoints.

Application *robustness* is the property of software that are able to survive consistently from data and code errors. This area is a major concern for complex numeric software that deal with data *uncertainties*. This is particularly the case for simulation applications [7].

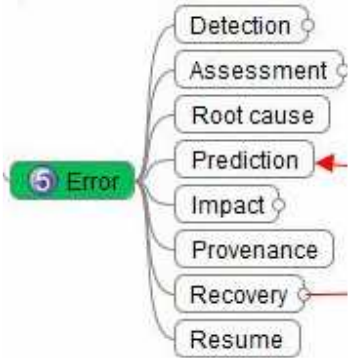

Figure 1. Error management.

G. Resilience

This is also a primary concern for the applications faced to system and hardware errors. In the following, we include both (application external) fault-tolerance and (internal) robustness in the generic term *resilience* [1].

Therefore we do not follow here the definition given in [17]: "*By definition a failure is the impact of an error itself caused by a fault*."

But we fully adhere to the following observation: "*the response to a failure or an error depends on the context and the specific sensitivity to faults of the usage scenarios, applications and algorithms*" [17].

II. Error Detection

A. Error Characterization

We address in this paper application errors, e.g., out of bounds data values, undefined parameters, execution time-outs, result discrepancies and unexpected values. We do not address communication, hardware and operating systems errors. We suppose that they are handled by the appropriate fault-tolerance sub-systems, which might automatically correct some of them or take appropriate corrective action, e.g., re-routing lost messages. We also suppose that these errors can be signaled to the application-level software by the appropriate raising of exceptions and posting of signals. Thus, the applications can take whatever actions are needed, e.g., re-executing tasks on other resources in case of network partition. This can be defined by the application designers and even by the application users at runtime.

The early characterization of errors is difficult because of the complex software stack involved in the execution of multi-discipline and multi-scale applications on clouds. The consequence is that errors might be detected long after the root cause that initiated them occurred. Also, the error observed might be a complex consequence of the root cause, possibly in a different software layer (Figure 2).

Similarly, the exact tracing and provenance data may be very hard to sort out, because the occurrence of the original fault may be hidden deep inside the software stack.

Without explicit data dependency information and real-time tracing of the components execution, the impacted components and associated results may be unknown. Hence there is a need for explicit dependency information [10].

B. Error Ranking

The ranking of errors is dependent on the application logic and semantics (e.g., default values usage). It is also dependent on the logics of each software layer composing the software stack. Some errors might be recoverable (unresolved address, resource unavailable…), some others not (network partition…). In each case, the actions to recover and resume differ: ignore, retry, reassign, suspend, abort...

In all cases, resilience requires the application to include four components:

- a monitoring component for early error detection,
- an effective decision system, for provenance and impact assessment,
- a low overhead checkpointing mechanism,
- an effective recovery mechanism.

Further, some errors might be undetected and transient. Without explicit data dependency information and real-time tracing of the components execution, the impacted components and associated results may be unknown. Hence there is a need for explicit dependency information between the component executing instances and between the corresponding result data [12].

III. Error Management

Many open issues are still the subject of active research concerning application resilience. The paradigm ranges from code and data duplication and migration, to the monitoring of application behavior, and this includes also quick correctness checks on partial data values, the design of error-aware algorithms, as well as hybrid checkpointing-message logging features (Figure 1). We focus here only on application errors. We do not address hardware, systems and communication errors. We suppose that these errors are fully treated by the

appropriate system components [15][22]. We further suppose that they can be signaled to the applications by some exception events. This allows handling the consequences of the errors, e.g., communication failures, by the appropriate application resilience sub-system.

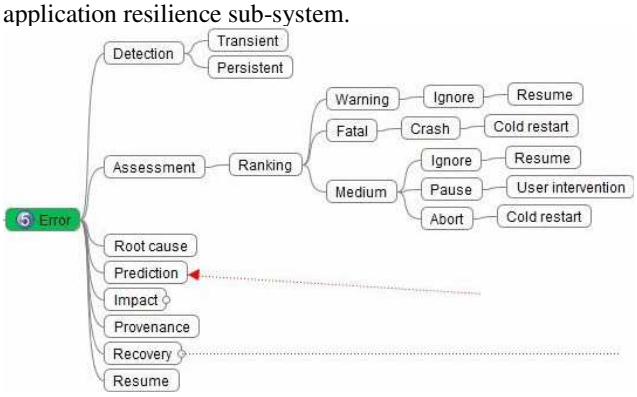

Figure 2. Error detection and assessment.

The baseline is:

- the early detection of errors
- root cause characterization
- characterization of transient vs. persistent errors
- the tracing and provenance of faulty data
- the identification of the impacted components and their associated corrupted results
- the ranking of the errors (warnings, fatal, medium) and associated actions (ignore, restart, backtrack)
- the identification of pending components
- the identification and purge of transient messages
- the secured termination of non-faulty components
- the secure storage of partial and consistent results
- the quick recovery of faulty and impacted components
- the re-synchronization of the components and their associated data
- the properly sequenced restart of the components

Each of these items needs appropriate implementation and algorithms in order to orchestrate the various actions required by the recovery of the faulty application components.

A. *Detection*

1) Anomaly Detection: Hardware and system fault-tolerance mechanisms can intercept errors [12]. Applications errors however must be explicitly taken into account in the code (Figure 3). This impacts severely the programming efforts and needs important design and re-programming efforts for existing codes [13].

2) Error Characterization: Similarly, error characterization is heavily dependent on the application logics [14]. It allows for error ranking, ranging from warnings to fatal. This is necessary to fine tune the fault-tolerance and resilience capabiities to the application and user requirements.

3) Root Cause Detection: root cause characterization and provenance information is the most difficult part in complex applications and systems. Most of the time, even sophisticated tracing mechanisms will fail to provide an accurate characterization of the multiple root causes that provoke errors and abnormal application behavior [17].

4) Impact Assessment: The next important step is the assessment of the error impact. This includes the impact on the subsequent tasks, on the data, and the evaluation of error propagations. Further, a domino effect is that the errors can impact the messages exchanged and in transit between tasks as well as the advent of the pending tasks. This is detailed in the following sections (B.1, B.2, C.1 and C.2)

B. *Impacted Tasks and Data*

1) Impacted Tasks: The application definition provides a detailed dependency relationship between tasks and data. It should therefore be straitghtforward to characterize the impacted tasks and data (Figure 4). However, the latency between error occurrence and their actual detection makes it difficult to precisely point out the exact time and location when an error occurred, particularly in distributed systems. Therefore, impacted tasks and data can bearly be defined without an undefined uncertainty. This paves the way for drastic backtracking policies and restarts. However, optimized checkpointing schemes, e.g., asymmetric [1], multi-level [18] and encoded checkpoints [22], alleviate somehow crude backtracking and checkpointing approaches by reducing their overhead, in both CPU and storage terms.

2) Corrupted Data: Similarly, corrupted data can originate from:

a) Application errors: Computation errors on correct data will produce erroneous results, e.g., speciification, algorithmic, programming erros. They can be spotted and corrected with unpredictable delays. Performance and overhead issues are Performance and overhead issues are not necessarily fundamental here because CPU and data demanding tasks might have to be backtracked and re-executed, incurring potentially very long delays.

b) Error propagation: Correct computations performed on previously polluted data may generate random errors on data processed subsequently. Errors cannot in this case be pointed out immediately, if at all. Restarting the application componenents from ancestor tasks might be a necessary option here. The exact and most accurate restart location may in some cases be difficult to characterize. Policy requirements and implementations are in this case the last resort.

C. *Impact*

1) Transient Messages: Transient messages are potentially emitted before a component failure. Identifying such data might be very difficult in distributed computing and collaborative codes. Indeed, failed tasks might have sent unknown numbers of messages and data to a potentially unknown number of descendant tasks, depending on the point of failure. Time-outs might here be necessary to consider transient messages to reach their destinations.

Purging all these messages is necessary to backtrack to a previous consistent checkpoint.

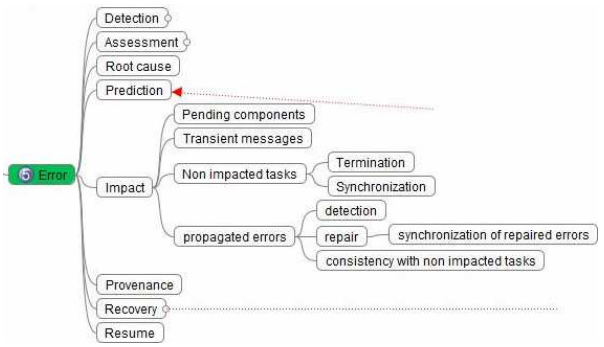

Figure 3. Error impact.

2) Pending Tasks: Pending tasks are in contrast easily characterized since they are waiting for incoming data or events raised by ancestor tasks. Pause and resuming of such tasks is an option, without systematically calling for their cold restart from a previous checkpoint. Opportunistic checkpoints might here be interesting to store already produced data and application state. This is related to asymmetric checkpoints [1], where the users define points of interest in the application runs where checkpoints and snapshots must be stored in order to prevent potential catastrophic failures later. So, CPU, storage and communication demanding tasks will in such cases be saved without the need to restart them later in case of application errors.

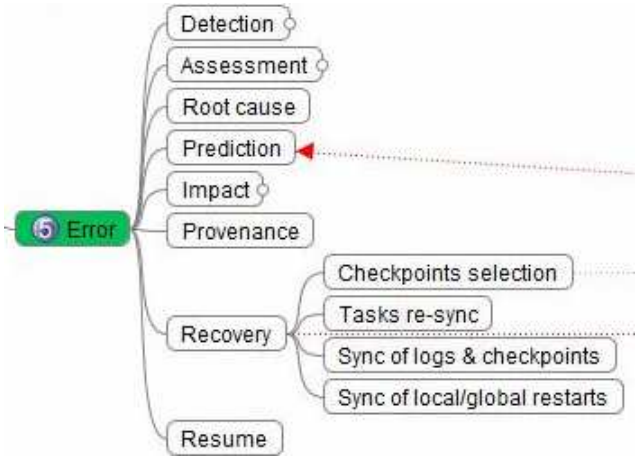

Figure 4. Error recovery.

D. Recovery

1) Termination of Non-faulty Tasks: As mentionned above, recovery of non-faulty tasks is straightforward if they are not directly linked to faulty tasks, or if they are explicitly waiting for incoming data or events. If they are directly linked to failed tasks, i.e., processing data produced by failed tasks, restarting them with the failed tasks may be necessary. Indeed, without a sophisticated control of the data exchanged between tasks, it my be impossible to characterize the subsets of data already processed correctly by subsequent tasks. This is also the case when using data pipelining between tasks. In this case, restarting the tasks from the beginning is necessary. Further, resuming the subsequent tasks also requires the ancestor failed tasks to restart at their adequate execution locations when failed. This is most of the time impossible in current systems. It requires repetitive incremental and partial checkpoints of state dta and produced results, which can have an important overhead (Figure 5).

2) Secured Storage of Non-faulty Data: The secured storage of non-faulty data is essential for the optimization of the recovery process. Although, if it does not succeed, backtracking to a preceding checkpoint in the execution run is an option.

3) Restart Selection: There might be several options available for a single coordinated restart or local partial restarts (Figure 6). Depending on the situation, ranging from warnings to erros and fatal ones, the distributed configuration of the applications might render a global coordinated restart unrealistic. Several partial local restarts might be preferable, and in all cases, less expensive in terms of CPU and resource consumptions (Section D.5, below).

4) Checkpoint Selection: An adequate checkpoint selection mechanism must be devised, which supports local restart in parallel and/or partial restarts from distributed and coordinated checkpoints. Here again, the versatility of the checkpointing mechanism, i.e., the support for multi-level checkpoints, is of first importance for reducing the restart overhead (Figure 7). But the cost is of course, the checkpointing overhead, both in terms of CPU and storage capacity, incurred. Encoding mechanisms, “shadowed” and “cloned” virtual disk images have been proposed to answer these concerns [23].

5) Coordinated Restart: Coordinated local restarts is a middle term option, between global cold restarts and multiple local restarts. As mentioned previously (Section D.3), a global coordinated cold restart is unrealistic in distributed systems because it requires stopping all tasks and restarting the whole application, which might require large computing resources and days of CPU time. Coordination is fundamental here and related to distributed computations. It follows that coordinated restarts must be implemented by a specific mechanism that selects timestamped data and checkpoints.

IV. IMPLEMENTATION

A. Overview

Several proposals have emerged recently dedicated to resilience and fault management in HPC systems [14][15][16].

The main components of such sub-systems are dedicated to the management of error, ranging from early error detections to error assessment, impact characterization, healing procedures concerning infected codes and data,

choice of appropriate steps backwards and effective low overhead restart procedures.

General approaches which encompass all these aspects are proposed for Linux systems, e.g., CIFTS [5]. More dedicated proposals focus on multi-level checkpointing and restart procedures to cope with memory hierarchy (RAM, SSD, HDD), hybrid CPU-GPU hardware, multi-core hardware topology and data encoding to optimize the overhead of checkpointing strategies, e.g., FTI [22]. The goal is to design and implement low overhead, high frequency and compact checkpointing schemes.

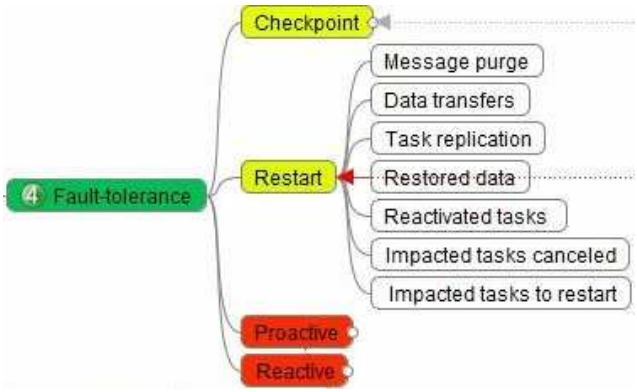

Figure 5. Restart.

Also, new approaches take benefit of virtualization technologies to optimize checkpointing mechanisms using virtual disks images on cloud computing infrastructures [23].

Two complementary aspects are considered:

- The detection and management of faults inherent to the hardware and software systems used
- The detection and management of faults emanating from the application code itself

Both aspects are different and imply different system components to react. However, unforeseen or incorrectly handled application errors may have undesirable effects on the execution of system components. The system and hardware fault management components might then have drastic procedure to confine the errors, which can lead to the application aborting. This is the case for out of bound parameter and data values, incorrect service invocations, if not correctly taken care of in the application codes.

This raises an important issue in algorithms design. Parallelization of numeric codes on HPC platforms is today taken into account in an expanding move towards petascale and future exascale computers. But so far, only limited algorithmic approaches take into account fault-tolerance from the start.

B. *Resilience Sub-system*

Generic system components have been designed and tested for fault-tolerance. They include fault-tolerance backpanes [5] and fault-tolerance interfaces [22]. Both target general procedures to cope with systematic monitoring of hardware, system and applications behaviors. Performance consideration limit the design options of such systems where incremental and multi-level checkpoints become the norm, in order to alleviate the overhead incurred by checkpoints storage and CPU usage. These can indeed exceed 25% of the total wall time requirements for scientific applications [22]. Other proposals take advantage of virtual machines technologies to optimize checkpoints storage using incremental ("shadowed" and "cloned") virtual disks images on virtual machines snapshots [23].

C. *Distributed Platform*

The distributed platform is built by the connection of two main components: 1) a workflow management system for application definition, deployment, execution and monitoring [2][9]; 2) a middleware allowing for distributed resource reservation, and execution of the applications on a wide-area network.

1) Workflow. The applications are defined using a workflow management system, i.e., YAWL [20]. This allows for dataflow and control flow specifications. It allows parameter definition and passing between application tasks. The tasks are defined incrementally and hierarchically. They can bear constraints that trigger appropriate code to cope with exceptions, i.e., exlets, and user-defined real-time runtime branchings. This allows for situational awareness at runtime and supports user interventions, when required. This is a powerful tool to deal with fault-tolerance and application resilience at runtime [1].

2) Middleware. The distribution of the platform is designed using an open-source middleware, i.e., Grid5000 [9]. This alows for reservation, deployment and execution of customized systems and application configurations. The Grid5000 nationwide infrastructure currently includes 12 sites in France and abroad, 19 research labs, 15 clusters, 1500 nodes, 8600 cores, connected by a 10Gb/s network. The resource reservations, deployment and running of the applications are made through standardized calls to specific system libraries. Because the infrastructure is shared between many research labs, resource reservation and job executions, i.e., applications, are queued with specific priority considerations.

3) Applications. Applications are defined, run and monitored using the standard workflow interface [6][19]. They invoke automatically or manually the tasks, as defined in the specification interface. Tasks in turn invoke the various executable components tranparently through the middleware, using Web services [21]. They are standard in YAWL and used to invoke executable codes specific to each

task. The codes are written in any programming language, ranging from Python to Java and C++. Script invocations with parameters are also possible. Parameter passing and data exchange between the executable codes are standardized in the workflow interface. Data structures are extendible user-defined templates to cope with all potential applications. As mentioned in the previous sections (Section "Workflow", above), constraints are defined and rules trigger component tasks based on data values, conditional checking. The testcases are distributed on a network of HPC clusters using the Grid5000 infrastructure. The hardware characteristics of the clusters are different. The application performance when running on various clusters are therefore different.

We use this infrastructure to deploy the application tasks on the various clusters and take advantage of the different cluster performance characteristics to benefit from load-balancing techniques combined with error management.

The automotive testcase presented here includes 17 different rear-mirror models tested for aerodynamics refinement. They are attached to a vehicle mesh of 22 million cells. A reference simulation was performed in 2 days on a 48 CPU non-distributed cluster with a total of 144 GB RAM. The result was a 2% drag reduction for the complete vehicle. The mesh will be eventually refined to include 35 million cells. A DES (Detached Eddy Simulation) flow simulation model is used.

The tasks include, from left to right in Figure 6:

- An initialization task for configuring the application (data files, optimization codes among which to choose…)
- A mesh generator producing the input data to the optimizer from a CAD file
- An optimizer producing the optimized data files (e.g., variable vectors)
- A partitioner that decomposes the input mesh into several sub-meshes for parallelization
- Each partition is input to a solver, several instances of which work on particular partitions
- A cost function evaluator, i.e., a drag evaluation
- A result gathering task for output and data visualization
- Error handlers in order to process the errors potentially raised by the solvers

The optimizer and solvers are all implemented using MPI. This allows highly parallel software executions. Combined with the parallelization made possible by the various mesh partitions, and the different geometry configurations of the testcase, it follows that there are three complementary parallelization levels in this testcase, which allow to fully benefit from the HPC clusters infrastructure.

Should an error occur during the solver processes, an exception is raised by the faulty tasks and they transfer the control to the corresponding error handler. This one will process the errors and trigger the appropriate actions, including:

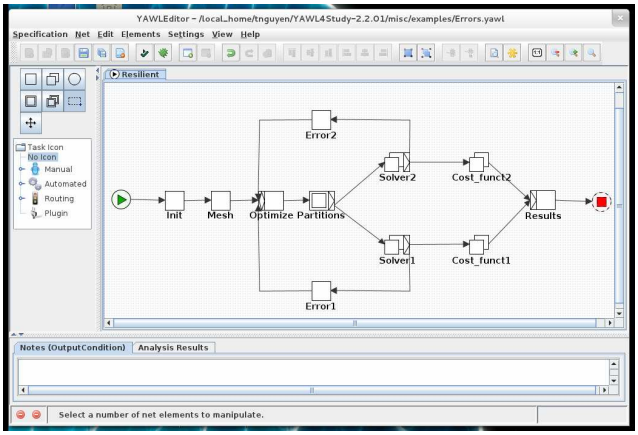

Figure 6. Distributed resilience testcase.

- Transfer automatically the solver task and data to another cluster, in case of CPU time limit or memory overflow: this is a load-balancing approach
- Retry the optimizer task with new input parameters requested from the user (number of iterations, switch optimizer code…)
- Ignore the error, if applicable, and resume the solver task

This approach merges two different and complementary techniques:

- Application-level error handling
- A load-balancing approach to take full benefit of the various cluster characteristics, for best application performance

Finally, the testbed implements the combination of a user-friendly workflow system with a grid computing infrastructure. It includes automatic load-balancing and resilience techniques. It therefore provides a powerful cloud infrastructure, compliant with the "Infrastructure as a Service" approach (IaaS).

V. CONCLUSION

High-performance computing and cloud infrastructures are today commonly used for running large-scale e-Science applications.

This has raised concerns about system fault-tolerance and application resilience. Because exascale computers are emerging, the need for supporting resilience becomes even more stringent.

Sophisticated and optimized functionalities are therefore required in the hardware, systems and application codes to support effectively error detection and recovery.

This paper defines concepts, details current issues and sketches solutions to support application resilience. Our

approach is currently implemented and tested on simulation testcases using a distributed platform that operates a workflow management system interfaced with a grid infrastructure, altogether providng a seamless cloud computing environment.

The platform provides functionalities for application specification, deployment, execution and monitoring. It features resilience capabilities to handle runtime errors. It implements the cloud computing "Infrastructure as a Service" paradigm using a user-friendly application workflow interface.

ACKNOWLEDGMENT

The authors wish to thank Alain Viari, Directeur de Recherche with INRIA, for many fruitful discussions concerning the testcase design and deployment.

This work is supported by the European Commission FP7 Cooperation Program "Transport (incl. aeronautics)", for the *GRAIN* Coordination and Support Action *("Greener Aeronautics International Networking"*), grant ACS0-GA-2010-266184.

It is also supported by the French National Research Agency ANR (*Agence Nationale de la Recherche*) for the OMD2 project (*Optimisation Multi-Discipline Distribuée)*, grant ANR-08-COSI-007, program COSINUS (*Conception et Simulation*).

REFERENCES

[1] T. Nguyên, L. Trifan and J-A Désidéri . "A Distributed Workflow Platform for Simulation". Proc. 4th Intl. Conf on Advanced Engineering Computing and Applications in Sciences (ADVCOMP2010). pp. 375-382. Florence (I). October 2010.

[2] E. Deelman et Y. Gil., "Managing Large-Scale Scientific Workflows in Distributed Environments: Experiences and Challenges", Proc. of the 2nd IEEE Intl. Conf. on e-Science and the Grid. pp. 165-172. Amsterdam (NL). December 2006.

[3] H. Simon. "Future directions in High-Performance Computing 2009-2018". Lecture given at the ParCFD 2009 Conference. Moffett Field (Ca). May 2009.

[4] Dongarra, P. Beckman et al. "The International Exascale Software Roadmap". Volume 25, Number 1, 2011, International Journal of High Performance Computer Applications, pp. 77-83. Available at: http://www.exascale.org/ Last accessed: 02/02/2012.

[5] R. Gupta, P. Beckman et al. "CIFTS: a Coordinated Infrastructure for Fault-Tolerant Systems", Proc. 38th Intl. Conf. Parallel Processing Systems. pp. 145-156. Vienna (Au). September 2009.

[6] D. Abramson, B. Bethwaite et al. "Embedding Optimization in Computational Science Workflows", Journal of Computational Science 1 (2010). Pp 41-47. Elsevier.

[7] A.Bachmann, M. Kunde, D. Seider and A. Schreiber, "Advances in Generalization and Decoupling of Software Parts in a Scientific Simulation Workflow System", Proc. 4th Intl. Conf. Advanced Engineering Computing and Applications in Sciences (ADVCOMP2010). Pp 247-258. Florence (I). October 2010.

[8] E.C. Joseph, et al. "A Strategic Agenda for European Leadership in Supercomputing: HPC 2020", IDC Final Report of the HPC Study for the DG Information Society of the EC. July 2010. Available at: http://www.hpcuserforum.com/EU/ Last accessed: 02/02/2012.

[9] T. Nguyên, L. Trifan, J.A. Désidéri. "A Workflow Platform for Simulation on Grids", Proc. 7th Intl. Conf. on Networking and Services (ICNS2011). pp. 295-302. Venice (I). May 2011.

[10] E. Sindrilaru, A. Costan and V. Cristea. "Fault-Tolerance and Recovery in Grid Workflow Mangement Systems", Proc. 4th Intl. Conf. on Complex, Intelligent and Software Intensive Systems. pp. 162-173. Krakow (PL). February 2010.

[11] The Apache Foundation. http://ode.apache.org/bpel-extensions.html #BPELExtensions-ActivityFailureandRecovery Last accessed: 02/02/2012.

[12] P. Beckman. "Facts and Speculations on Exascale: Revolution or Evolution?", Keynote Lecture. Proc. 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 135-142. Bordeaux (F). August 2011.

[13] P. Kovatch, M. Ezell, R. Braby. "The Malthusian Catastrophe is Upon Us! Are the Largest HPC Machines Ever Up?", Proc. Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 255-262. Bordeaux (F). August 2011.

[14] R. Riesen, K. Ferreira, M. Ruiz Varela, M. Taufer, A. Rodrigues. "Simulating Application Resilience at Exascale", Proc. Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 417-425. Bordeaux (F). August 2011.

[15] P. Bridges, et al. "Cooperative Application/OS DRAM Fault Recovery", Proc. Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). pp. 213-222. Bordeaux (F). August 2011.

[16] Proc. 5th Workshop INRIA-Illinois Joint Laboratory on Petascale Computing. Grenoble (F). June 2011. http://jointlab.ncsa.illinois.edu/events/workshop5/ Last accessed 02/02/2012.

[17] F. Capello, et al. "Toward Exascale Resilience", Technical Report TR-JLPC-09-01. INRIA-Illinois Joint Laboratory on PetaScale Computing. Chicago (Il.). 2009. http://jointlab.ncsa.illinois.edu/

[18] Moody A., G.Bronevetsky, K. Mohror, B. de Supinski. Design, "Modeling and evaluation of a Scalable Multi-level checkpointing System", Proc. ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC10). pp. 73-86. New Orleans (La.). Nov. 2010. http://library-ext.llnl.gov Also Tech. Report LLNL-TR-440491. July 2010. Last accessed: 02/02/2012.

[19] Adams M., ter Hofstede A., La Rosa M. "Open source software for workflow management: the case of YAWL", IEEE Software. 28(3): 16-19. pp. 211-219. May/June 2011.

[20] Russell N., ter Hofstede A. "Surmounting BPM challenges: the YAWL story.", Special Issue Paper on Research and Development on Flexible Process Aware Information Systems. Computer Science. 23(2): 67-79. pp. 123-132. March 2009. Springer 2009.

[21] Lachlan A., van der Aalst W., Dumas M., ter Hofstede A. "Dimensions of coupling in middleware", Concurrency and Computation: Practice and Experience. 21(18):233-2269. pp. 75-82. J. Wiley & Sons 2009.

[22] Bautista-Gomez L., et al., "FTI: high-performance Fault Tolerance Interface for hybrid systems", Proc. ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC11), pp. 239-248, Seattle (Wa.)., November 2011.

[23] Nicolae B. and Cappello F., "BlobCR: Efficient Checkpoint-Retart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots", Proc. ACM/IEEE Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), pp. 145-156, Seattle (Wa.)., November 2011.