Resilience for Collaborative Applications on Clouds

Toan Nguyen, Jean-Antoine Desideri

Resilience for Collaborative Applications on Clouds

Fault-Tolerance for distributed HPC applications

Toàn Nguyên and Jean-Antoine Désidéri

Project OPALE
INRIA
38334 Saint-Ismier, France

{Toan.Nguyen, Jean-Antoine.Desideri}@inria.fr

Abstract. Because e-Science applications are data intensive and require long execution runs, it is important that they feature fault-tolerance mechanisms. Cloud and grid computing infrastructures often support system and network fault-tolerance. They repair and prevent communication and software errors. They allow also checkpointing of applications, duplication of jobs and data to prevent catastrophic hardware failures. However, only preliminary work has been done so far on application resilience, i.e., the ability to resume normal execution following application errors and abnormal executions. This paper is an overview of open issues and solutions for such errors detection and management. It also overviews the implementation of a workflow management system to design, deploy, execute, monitor, restart and resume distributed HPC applications on cloud infrastructures in cases of failures.

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

1 Introduction

Scientific applications are required today to design, simulate, optimize and manufacture artifacts, ranging from nanotubes to electronic devices and cruisers to airliners.

In order to design quickly these artifacts, long running simulations are executed. For example, multi-discipline scenarios are implemented, where hydraulic, thermic, fluid and electromagnetic simulation software collaborate for the design of nuclear plants.

Long running executions lasting days and even weeks on large HPC clusters suffer from reliability problems concerning the hardware and software infrastructures

[15][17]. Fault-tolerance mechanisms are therefore required. They tend to be multi-level, each aspect corresponding to a different level with its specific sources of errors: network communications, distributed middleware, operating systems, collaborating application codes.

The efforts on application errors tend to be the focus of active research today. This is due in part to optimization concerns, and to another part for fault-tolerance concerns [16].

Optimization concerns target the speed-up of CPU and data intensive demanding applications [3]. Parallelization techniques take advantage today of multi-core super-computers. However the 100K+ multi-core HPC clusters today are error-prone and their mean-time between failures is in the order of minutes [13]. Therefore, effective and low-overhead fault-tolerance application algorithms and codes are necessary.

Further, applications misbehavior and errors have multiple origins, which are not necessarily programming errors. They might originate in unforeseen data configurations, especially in simulation applications, unexpected data values, unpredictable behaviors in case of multiple errors cumulating abnormalities, etc.

Important efforts are required to handle these complex abnormal application situations [4][8][14].

This paper addresses the management of application errors and abnormal behavior. It defines terms (section 2), addresses open issues and solutions for error detection (Section 3) and error management (Section 4). It sketches also implementation issues using a workflow management system (Section 5). Section 6 is a conclusion.

A prototype system based on a distributed workflow platform for the design, deployment, execution and monitoring of HPC applications is briefly described. The platform features resilience capabilities to address the application runtime errors.

2 Definitions

Because many terms are used in the fault-tolerance area, we give in this section a definition of various terms used in the domain and pave the way for an ontology of the required concepts.

An interesting definition of errors, faults and failures is given in a system such as Apache's ODE [11], system failures and application faults address different types of errors.

2.1 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).

2.2 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].

2.3 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.

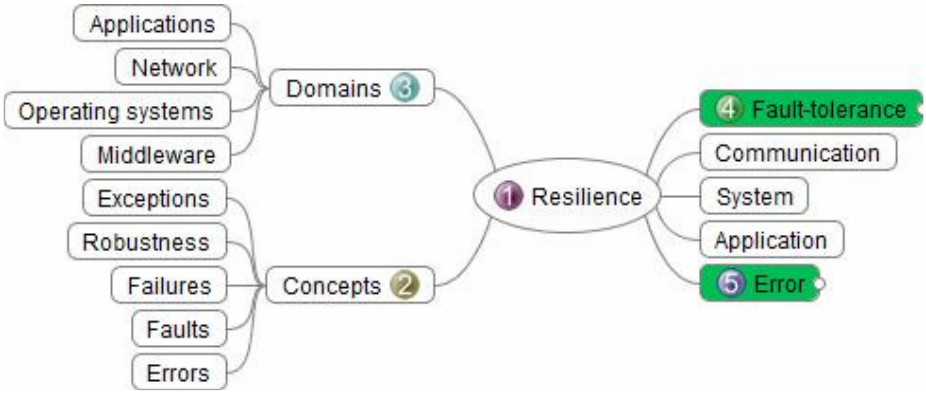

Fig. 1. Resilience domains and concepts.

2.4 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.

2.5 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.

2.6 Recovery

We assume therefore that the *recovery* procedures 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 using previously stored checkpoints.

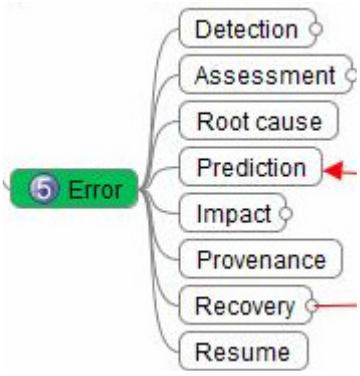

Fig. 2. Error management.

2.7 Resilience

Application *resilience* is the property of software that are able to survive consistently from data and code errors (Figure 2). This area is a major concern for complex numeric software that deal with data uncertainties. This is particularly the case for simulation applications [7].

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].

3 Error Detection

3.1 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, out-of-memory execution, etc. 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.

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].

3.2 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,
- a (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].

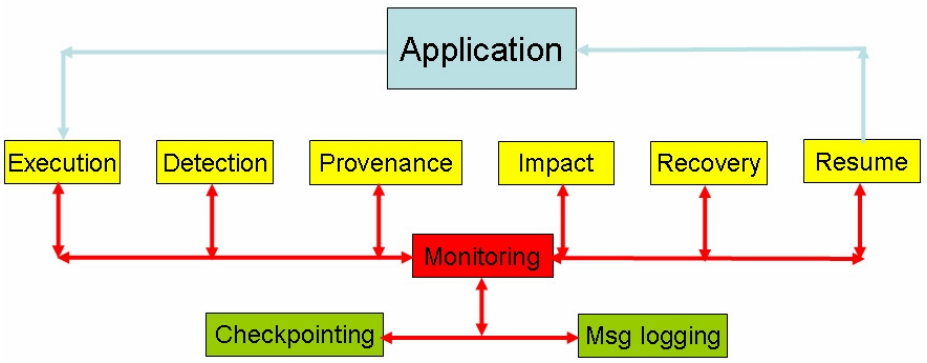

Fig. 3. Resilience sub-system.

3.3 Resilience Sub-system

A sub-system dedicated to application resilience includes therefore several components in charge of specific tasks contributing to the management of errors and consistent resuming of the applications (Figure 3). First, it includes an intelligence engine in charge of the application monitoring and of the orchestration of the resilience components [5]. This engine runs as a background process in charge of event listening during the execution of the applications. It is also in charge of triggering the periodic checkpointing mechanism, depending on the policy defined for the applications being monitored [22]. It is also in charge of triggering the message-logging component for safekeeping the messages exchanged between tasks during their execution. This component is however optional, depending on the algorithms implemented, e.g., checkpointing only or hybrid checkpoint-message logging approaches. Both run as background processes and should execute without user intervention. Should an error occur, an error detection component that is constantly listening to the events published by the application tasks and the operating system raises the appropriate exceptions to the monitoring component. The following components are then triggered in such error cases: an optional provenance component which is in charge of root cause characterization, whenever possible. An impact assessment component is then triggered to evaluate the consequences of the error on the application tasks and data, that may be impacted by the error. Next, a recovery component is triggered in charge of restoring the impacted tasks and the associated data, in order to re-synchronize the tasks and data, and restore the application to a previous consistent state. A resuming component

is finally triggered to deploy and rerun the appropriate tasks and data on the computing resources, in order to resume the application execution. In contrast with approaches designed for global fault-tolerance systems, e.g., CIFTS [15], this functional architecture describes a sub-system dedicated to application resilience. It can be immersed in, or contribute to, a more global fault-tolerance system that includes also the management of system and communication errors.

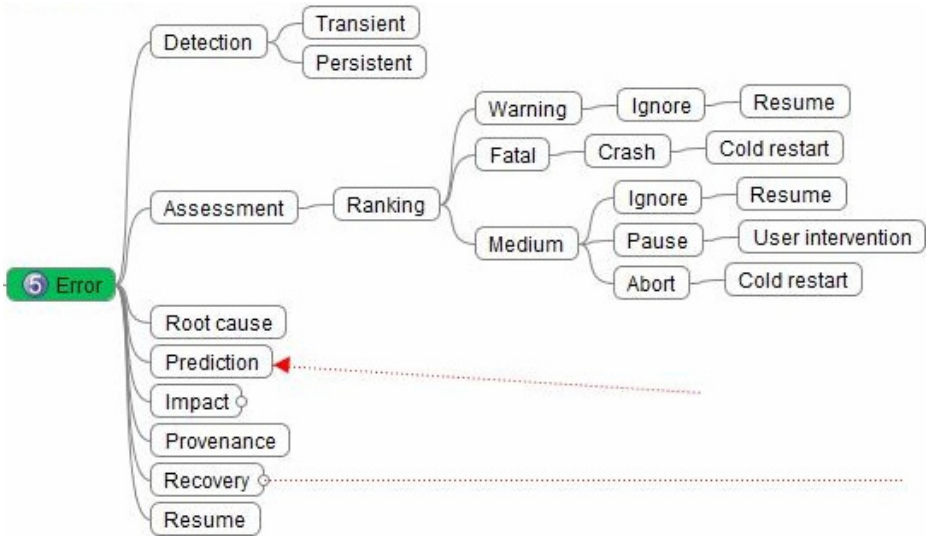

Fig. 4. Error detection and assessment.

4 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 3). 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 (Section 3.3).

The baseline is (Figure 4):

- 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.

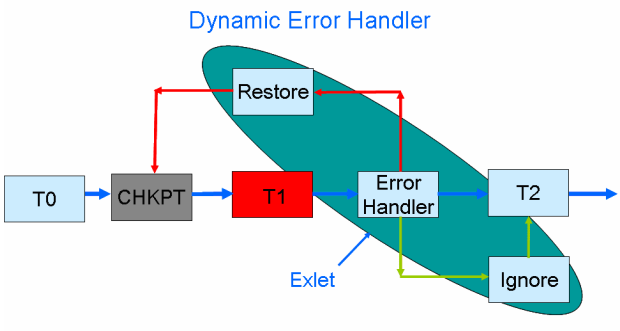

Fig. 5. Error handler.

4.1 Detection

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

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 capabilities to the application and user requirements.

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].

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 (4.2, 4.3).

4.2 Impacted Tasks and Data

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. 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.

Corrupted Data. Similarly, corrupted data can originate from application errors and from error propagation (Figure 6):

Application errors. Computation errors on correct data will produce erroneous results, e.g., specification, 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.

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.

4.3 Impact

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 mes-

sages 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.

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.

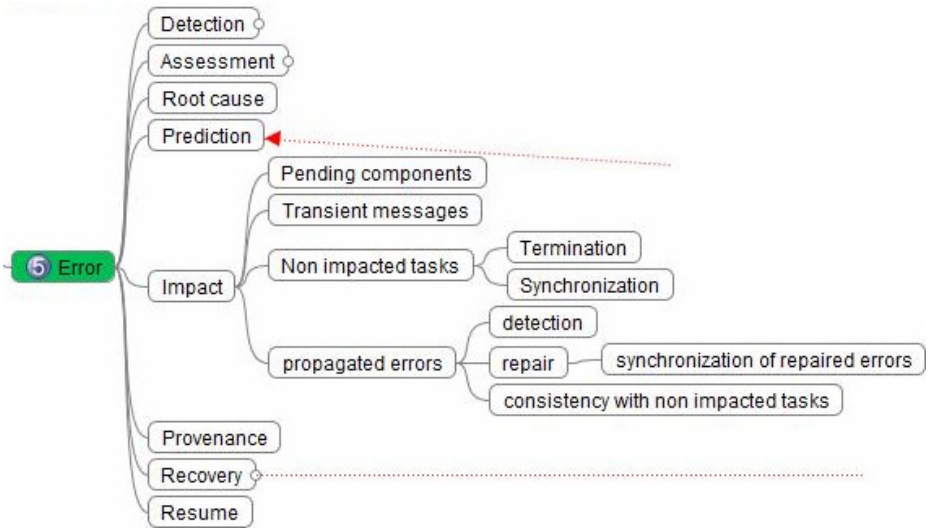

Fig. 6. Error impact.

4.4 Recovery

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 may 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 7).

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.

Restart Selection. There might be several options available for a single coordinated restart or local partial restarts (Figure 8). 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 "Coordinated Restart", below).

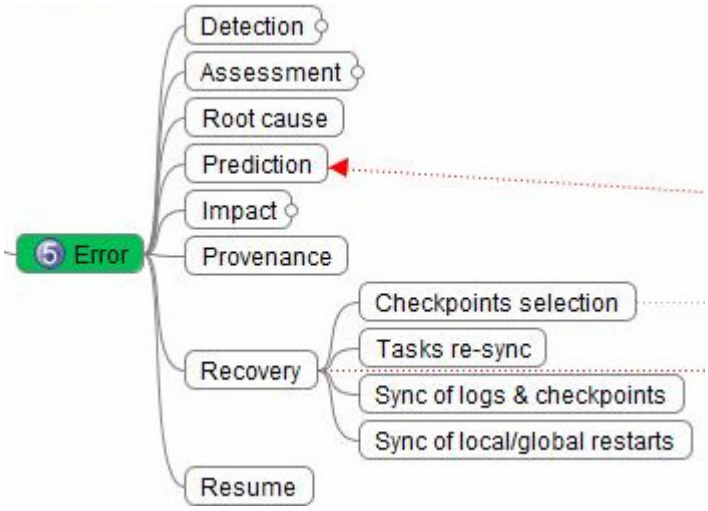

Fig. 7. Error recovery.

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 9). 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].

Coordinated Restart. Coordinated local restarts is a middle term option, between global cold restarts and multiple local restarts. As mentioned previously (Section 4.3), 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.Italso requires the careful selection of those checkpoints strictly needed for the restart. The latter will execute local restarts with the appropriate checkpoints selected.

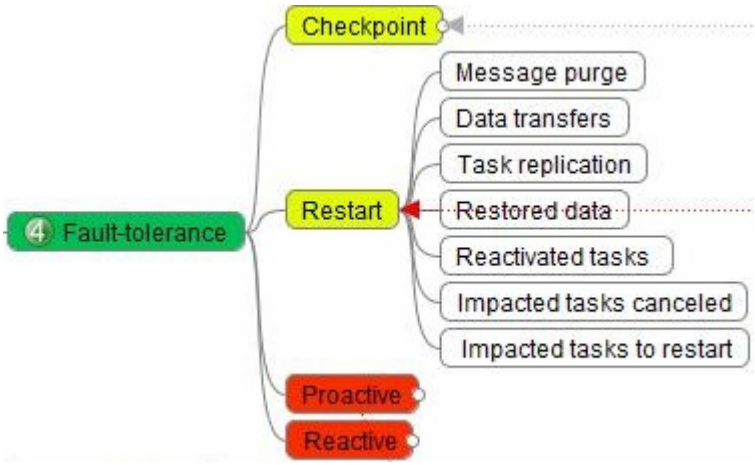

Fig. 8. Restart.

5 Implementation

5.1 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.

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.

5.2 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].

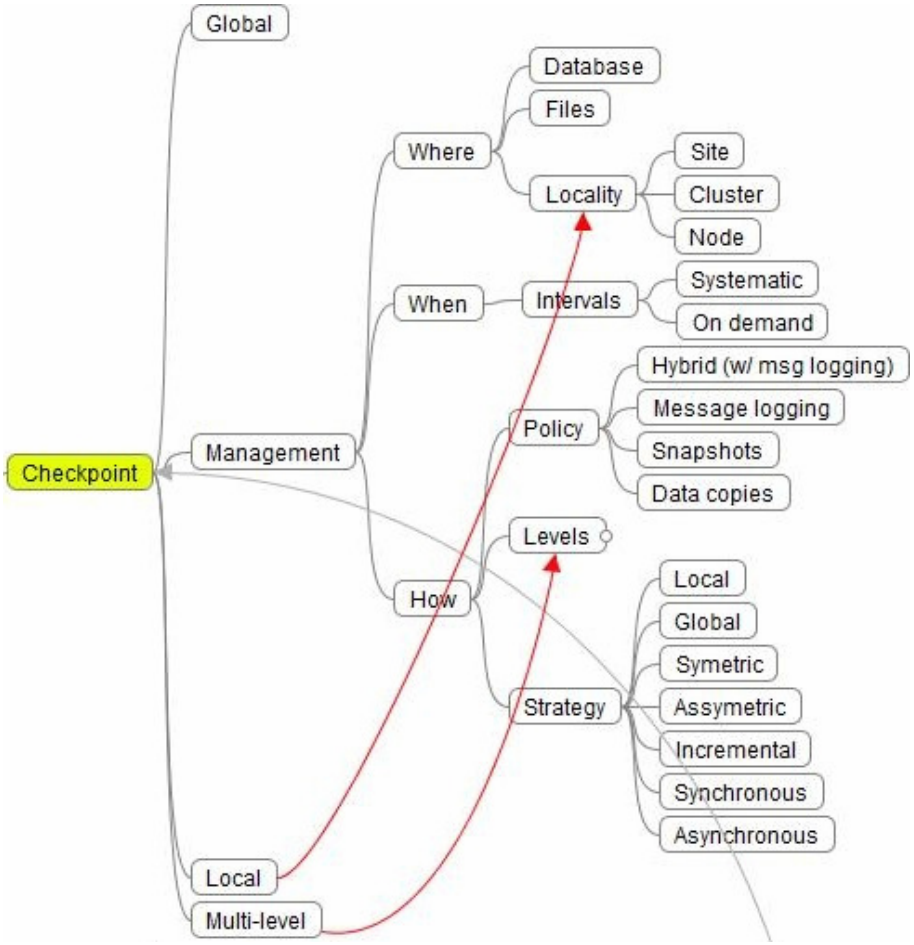

Fig. 9. Checkpoints.

6 Conclusion

The advent of petascale computers has raised concerns about system fault-tolerance and application resilience. Because exascale computers are now emerging, these concerns become even more stringent.

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

Large-scale applications also require distributed and heterogeneous environments to run collaborative multidiscipline projects. Workflow management systems are good candidate to deploy and control these applications because they require high-level and non-expert level dynamic features, e.g., interactive control and visualization.

They also require dynamic reconfiguration capabilities, e.g., in case of task redeployment because of hardware and software failures. Overall, they require resilience support because of potential software errors and erratic and unexpected behavior, e.g., because of wrong simulation parameters.

This paper defines concepts, details current problems and addresses solutions to application resilience. This approach is currently implemented and tested on simulation testcases using a distributed platform that operates a workflow management system interfaced with a cloud infrastructure. An automotive testcase is presented addressing a vehicle rear-mirror drag optimization.

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

Acknowledgments

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. Nguyên, T.,Trifan, L., Désidéri, J-A. : A Distributed Workflow Platform for Simulation. In: 4th Intl. Conf on Advanced Engineering Computing and Applications in Sciences,pp. 375-382. IARIA (2010)
2. Deelman, E. , Gil, Y.: Managing Large-Scale Scientific Workflows in Distributed Environments: Experiences and Challenges. In: 2nd IEEE Intl. Conf. on e-Science and the Grid.,pp. 165-172. IEEE, New York (2006)
3. Simon, H.: Future directions in High-Performance Computing 2009-2018. Lecture given at the ParCFD 2009 Conference (2009)
4. Dongarra, J., Beckman, P., 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/`
5. Gupta, R. , Beckman, P., et al. : CIFTS: a Coordinated Infrastructure for Fault-Tolerant Systems. In: 38th Intl. Conf. Parallel Processing Systems., pp. 145-156. (2009)
6. Abramson, D., Bethwaite, B., et al.: Embedding Optimization in Computational Science Workflows, Journal of Computational Science 1 (2010), pp. 41-47. Elsevier.
7. Bachmann, A., Kunde, M. , Seider, D., Schreiber, A.: Advances in Generalization and Decoupling of Software Parts in a Scientific Simulation Workflow System. In: 4th Intl. Conf. Advanced Engineering Computing and Applications in Sciences p 247-258. (2010)

8. Joseph, E.C. , 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/
9. Nguyên, T.,Trifan, L., Désidéri, J-A. : A Workflow Platform for Simulation on Grids. In: 7th Intl. Conf. on Networking and Services (ICNS2011), pp. 295-302 (2011)
10. Sindrilaru, E., Costan, A., Cristea, V.: Fault-Tolerance and Recovery in Grid Workflow Mangement Systems. In: 4th Intl. Conf. on Complex, Intelligent and Software Intensive Systems. LNCS,Vol. 6853, pp 163-173. Springer, Heidelberg (2011)
11. The Apache Foundation. http://ode.apache.org/bpel-extensions.html#BPELExtensions-ActivityFailureandRecovery
12. Beckman, P.: Facts and Speculations on Exascale: Revolution or Evolution? Keynote Lecture. In: 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011), . LNCS,Vol. 6853, pp. 135-142. Springer, Heidelberg (2011)
13. Kovatch, P., Ezell, M., Braby, R.: The Malthusian Catastrophe is Upon Us! Are the Largest HPC Machines Ever Up? In: Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011). LNCS,Vol. 6853, pp. 255-262. Springer, Heidelberg (2011)
14. Riesen, R., Ferreira, K., Ruiz, M., Varela, Taufer, M., Rodrigues, A.: Simulating Application Resilience at Exascale. In: Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011), LNCS,Vol. 6853, pp. 417-425. Springer, Heidelberg (2011)
15. Bridges, P., et al: Cooperative Application/OS DRAM Fault Recovery. In: Resilience Workshop at 17th European Conf. Parallel and Distributed Computing (Euro-Par 2011), pp. 213-222. Springer, Heidelberg (2011)
16. Proc. 5th Workshop INRIA-Illinois Joint Laboratory on Petascale Computing. Grenoble (F). June 2011. http://jointlab.ncsa.illinois.edu/events/workshop5
17. Capello, F.: 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., Bronevetsky, G., Mohror, K. , de Supinski. B.: Design, Modeling and evaluation of a Scalable Multi-level checkpointing System. In: ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC10), pp. 73-86. New Orleans (La.). (2010) http://library-ext.llnl.gov Also Tech. Report LLNL-TR-440491. 2010.
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. (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. 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), pp. 75-82. J. Wiley & Sons (2009)
22. Bautista-Gomez, L., et al., FTI: high-performance Fault Tolerance Interface for hybrid systems. In: ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC11), pp. 239-248, Seattle (Wa.). (2011)
23. Nicolae, B., Cappello F.: BlobCR: Efficient Checkpoint-Retart for HPC Applications on IaaS Clouds using Virtual Disk Image Snapshots. In: ACM/IEEE Intl. Conf. High Performance Computing, Networking, Storage and Analysis (SC11), pp. 145-156, Seattle (Wa.), 2011