Performance comparison of hierarchical checkpoint protocols grid computing

Ndeye Massata NDIAYE[1,2], Pierre SENS[1], and Ousmane THIARE[2]

(1) Regal team, LIP6, UPMC Paris Jussieu France

(2) Gaston Berger University of Saint-Louis Senegal

***Abstract* — Grid infrastructure is a large set of nodes geographically distributed and connected by a communication. In this context, fault tolerance is a necessity imposed by the distribution that poses a number of problems related to the heterogeneity of hardware, operating systems, networks, middleware, applications, the dynamic resource, the scalability, the lack of common memory, the lack of a common clock, the asynchronous communication between processes. To improve the robustness of supercomputing applications in the presence of failures, many techniques have been developed to provide resistance to these faults of the system. Fault tolerance is intended to allow the system to provide service as specified in spite of occurrences of faults. It appears as an indispensable element in distributed systems. To meet this need, several techniques have been proposed in the literature. We will study the protocols based on rollback recovery. These protocols are classified into two categories: coordinated checkpointing and rollback protocols and log-based independent checkpointing protocols or message logging protocols. However, the performance of a protocol depends on the characteristics of the system, network and applications running. Faced with the constraints of large-scale environments, many of algorithms of the literature showed inadequate. Given an application environment and a system, it is not easy to identify the recovery protocol that is most appropriate for a cluster or hierarchical environment, like grid computing. While some protocols have been used successfully in small scale, they are not suitable for use in large scale. Hence there is a need to implement these protocols in a hierarchical fashion to compare their performance in grid computing. In this paper, we propose hierarchical version of four well-known protocols. We have implemented and compare the performance of these protocols in clusters and grid computing using the Omnet++ simulator.**

***Keywords* — Grid computing, fault tolerance, checkpointing, message-logging**

I. Introduction

Molecular biology, astrophysics, high energy physics, those are only a few examples among the numerous research fields that have needs for tremendous computing power, in order to execute simulations, or analyze data. Increasing the computing power of the machines to deal with this endlessly increasing needs has its limits. The natural evolution was to divide the work among several processing units. Parallelism was first introduced with monolithic parallel machines, but the arrival of high-speed networks, and especially Wide Area Network (WAN) made possible the concept of clusters of machines, which were further extended to large scale distributed platforms, leading to a new field in computer science, grid computing.

The first definition of a grid has been given by Foster and Kesselman in [40]. A grid is a distributed platform which is the aggregation of heterogeneous resources. They do an analogy with the electrical power grid. The computing power provided by a grid should be transparently made available from everywhere, and for everyone. The ultimate purpose is to provide to scientific communities, governments and industries an unlimited computing power, in a transparent manner. This raised lots of research challenges, due to the complexity of the infrastructure. Heterogeneity is present at all levels, from the hardware (computing power, available memory, interconnection network), to the software (operating system, available libraries and software), via the administration policies.

From this definition, several kinds of architectures were born. One of the most commonly used architecture, referred to as remote cluster computing, is composed of the aggregation of many networked loosely coupled computers, usually those computers are grouped into clusters of homogeneous and well connected machines. These infrastructures are often dedicated to scientific or industrial needs, and thus provide large amount of computing resources, and a quite good stability.

Today, grid computing technologies make it possible to securely share data and programs for multiple computers, whether desktop or personal supercomputers. These resources are networked and shared through software solutions. In recent years, grid technology has emerged as an important tool for solving compute-intensive problems within the scientific community and in industry. To further the development and adoption of this technology, researchers and practitioners from different disciplines have collaborated to produce standard specifications for creating large-scale, interoperable grid system. The focus of this activity has been the Open Grid Forum (OGF) [8], but other standard development organizations have also produced specifications, such as

[9][10], that are used in grid systems. To fully transition grid technology to operational use and to expand the range and scale of grid applications, grid systems must exhibit high reliability; i.e. they must be able to continuously provide correct service [11]. Moreover, it is important that the specifications used to build these systems fully support reliable grid services. With the increase in use of grid technology, achieving these goals will be made more difficult as grid systems become larger, more heterogeneous in composition, and more dynamic. Many grids are appearing in the sciences, production grids are now being implemented in companies and among agencies: Grid'5000, TeraGrid, Sun Grid, Xgrid ... Grid computing will allow dynamic sharing of resources among participants, organizations and businesses in order to be able to pool, and thus run compute-intensive applications or treatment of very large volumes of data.

Since the failure probability increases with a rising number of components, fault tolerance is an essential characteristic of massively parallel systems. Such systems must provide redundancy and mechanisms to detect and localize errors as well as to reconfigure the system and to recover from error states. A fault tolerant approach may therefore be useful in order to potentially prevent a faulty node affecting the overall performance of the application. Fault tolerance appears as an indispensable element in grid computing. Many protocols for distributed computing have been designed [1]. These protocols are classified into four different classes, namely, coordinated checkpointing, communication induced checkpointing, independent checkpointing and log-based protocols.

We have implemented and compare the performance of these protocols in clusters and grid computing using the Omnet++ simulator [7].

Section II describes the protocols implemented in Omnet++. In section III, we talk about hierarchical checkpointing for grids. The experimental setup and results obtained by executing these protocols are presented in Section IV. In section V, we present the related work and finally section VI concludes.

II. CHECKPOINT AND ROLLBACK-RECOVERY PROTOCOLS

Checkpointing is a standard method for the repair of faults in systems. The idea is to save the state of the system on a stable periodic to prevent breakdowns (Fig. 1). That way when you restart after a power failure, the state saved newest restored and execution resumes its course before the crash. The overall status of a distributed system is defined by the union of local states of all processes belonging to the system.

Taking checkpoints is the process of periodically saving the state of a running process to durable storage. Checkpointing allows a process that fails to be restarted from the point its state was last saved, or its *checkpoint*. If the host processor has not failed, temporal redundancy can be used to *roll back* and restart the process on the same platform. As in other systems, this method is widely used in grids [36][37][38]. Otherwise, if the host has failed, the process may be *migrated*, or transferred, to a different execution environment where it can be restarted from a checkpoint (a technique also referred to as *failover*). This section begins by discussing checkpoint and process migration methods used in commercial and science grid systems that are based on methods used in high-performance cluster computing. This is followed by discussion of new methods being developed or adapted for scaled grid environments, together with related issues that need to be resolved. Most notable is the issue of finding efficient methods for checkpointing many concurrent, intercommunicating processes, so that in the event of failure, they can resume from a common saved state [39]. Checkpointing can be initiated either from within grid systems or within applications.

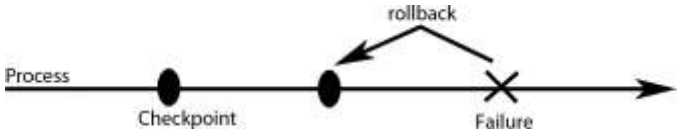

Fig 1: Rollback-Recovery

There are two main classes of protocols: coordinated checkpointing and message logging.

A. Coordinated checkpointing

Coordinated checkpointing is an attractive approach for transparently adding fault tolerance to distributed applications without requiring additional programmer ef- forts. In this approach, the state of each process in the sys- tem is periodically saved on stable storage, which is called a checkpoint of the process. To recover from a failure, the system restarts its execution from a previous error-free, consistent global state recorded by the checkpoints of all processes. More specifically, the failed processes are re-started on any available machine and their address spaces are restored from their latest checkpoints on stable storage. Other processes may have to rollback to their checkpoints on stable storage in order to restore the entire system to a consistent state. Coordinated checkpointing simplifies failure recovery and eliminates domino effects in case of failures by preserving a consistent global checkpoint on stable storage. However, the approach suffers from high overhead associated with the checkpointing process. Two approaches are used to reduce the overhead: First is to minimize the number of synchronization messages and the number of checkpoints, the other is to make the checkpointing process nonblocking.

The protocol requires processes coordinate their checkpoints to form a consistent global state. A global state is consistent if it does not include any orphan messages (i.e, a message received but not already sent). This approach simplifies the recovery and avoids the domino effect, since every process always restarts at the resume point later. Also, the protocol requires each process to maintain only one permanent checkpoint in stable storage, reducing the overhead due to storage and release of checkpoints (garbage collection) [1].

Its main drawback however is the large latency that require interaction with the outside world, in this case the solution is to perform a checkpoint after every input / output. To improve the performance of the backup coordinated, several techniques have been proposed. We have implemented as non-blocking coordinated checkpointing.

1) *Non-blocking coordinated checkpointing*

A nonblocking checkpointing algorithm does not require any process to suspend its underlying computation. When processes do not suspend their computations, it is possible for a process to receive a computation message from an other process which is already running in a new checkpoint interval. If this situation is not properly dealt with, it may result in an inconsistency. For example, in Fig. 2, *P*2 initiates a checkpointing process. The example of coordinated checkpoint non-blocking is that of Chandy and Lamport algorithm [2]. This algorithm uses markers to coordinate the backup, and operates under the assumption of FIFO channels. In [3], a comparison of protocols for coordinated checkpoint blocking and non-blocking has been made. Experiments have shown that the synchronization between nodes induced by the protocol blocking further penalize the performance of the calculation with a non-blocking protocol. However, using frequencies of taken checkpoints usual performance of the blocking approach is better on a cluster to high-performance communications.

2) *Communication induced checkpointing*

This protocol defines two types of checkpoints [1]: local checkpoints taken by processes independently, to avoid the synchronization of coordinated backup and forced checkpoints based on messages sent and received and dependency information carried 'piggyback' on these posts, so to avoid the domino effect of uncoordinated backup, ensuring the advancement of online collection. Unlike coordinated checkpoint protocols, the additional cost due to the medium access protocol disappears because the protocol does not require any message exchange to force a checkpoint: this information is inserted piggyback on the messages exchanged.

B. Message-Logging protocols

Message logging (for example [12] [13] [14] [15] [16] [17] [18] [19] [20]) is a common technique used to build systems that can tolerate process crash failure. These protocols required that each process periodically record its local state and log the messages it received after having recorded that state. When a process crashes, a new process is created in its place: the new process is given the appropriate recorded local state, and then it is sent the logged messages in the order they were originally received. Thus, message logging protocols implement an abstraction of a resilient process in which the crash of a process is translated into intermittent unavailability of that process.

All message logging protocols require that the state of a recovered process be consistent with the states of the other processes. This consistency requirement is usually expressed in terms of *orphan processes*, which are surviving processes whose states are inconsistent with the recovered state of crashed process. Thus, in the terminology of message logging, message logging protocols must guarantee that there are no orphan processes, either through careful logging of through a somewhat complex recovery protocol.

The logging mechanism uses the fact that a process can be modeled as a sequence of deterministic state intervals, each event begins with a non-deterministic. An event may be receiving a message, or issued or other event in the process. It is deterministic if from a given initial state, it always happens at the same final state. [1]

The principle of Logging is to record on a reliable storage any occurrences of non-deterministic events to be able to replay them in recovering from a failure. During execution, each process performs periodic backups of their states, and recorded in a log information about messages exchanged between processes. There are three message-logging categories: optimistic, pessimistic and causal.

1) *Pessimistic message-logging*

This protocol was designed under the assumption that a failure may occur after any nondeterministic event (i.e. message reception). Then, each message is saved on a stable storage before to be delivering to the application.

These protocols are often made reference to the synchronized because when logging process logs an event of non-deterministic stable memory, it waits for an acknowledgment to continue its execution.

In a pessimistic logging system, the status of each process can be recovered independently. This property has four advantages:

- Process can send messages to the outside without using a special protocol
- The process restarted at the most recent checkpoint.
- Recovery is simple because the effects of a failure are limited only on the fail process
- The garbage collector is simple

The main drawback is the high latency of communications, which results in degradation of the applications response time. Several approaches have been developed to minimize synchronizations:

- The use of semiconductor memories such as non-volatile stable support
- The sender based message logging (SBML) [14] which preserves the determinant or the message in the volatile memory of the transmitter, instead of a remote memory

2) *Optimistic message-logging*

This protocol uses the assumption that the logging of a message on reliable support will be complete before a failure occurs. Indeed, during the process execution, the determinants of messages are stored in volatile memory, before being saved periodically on stable support. The storage stable memory is asynchronous: the protocol does not require the application to be blocked during the backup memory stable. Induced latency is then very low.

However, a failure may occur before the messages are saved on stable storage. In this case, the information stored in volatile memory of the process down is lost and the messages sent by this process are orphaned. This can produce a domino effect of rollbacks, which increases the recovery time.

3) *Causal message-logging*

This protocol combines the advantages of both previous methods. As optimistic logging, it avoids the synchronized access to stable, except during the input / output. As pessimistic logging, it allows the process to make interactions with the outside world independently, and does not create process orphan. Causal logging protocols piggyback determinants of messages previously received on outgoing messages so that they are stored by their receivers.

III. Hierarchical checkpointing for Grids

The architecture of a grid can be defined as a set of clusters connected by a WAN-type network. The cluster consists of multiple nodes connected by a broadband network. We adopt a hierarchical scheme. In each cluster, there is one leader connected to all other nodes of its cluster. All leaders are connected together (Fig. 2).

The leader assumes the role of intermediary in the inter-cluster communications. The backup takes place in four phases:

1) *Initialization*: an initiator sends a checkpoint-request to its leader
2) *Coordination of leaders*: the leader transfers the checkpoint request to the other leaders
3) *Local checkpointing* : Each leader initiates a checkpoint in its cluster
4) *Termination*: When local checkpoint is over, each leader sends an acknowledgement to the initial leader.

The recovery follows the same rules as the backup: coordination phase of the leaders, and a phase of recovery limited to the cluster.

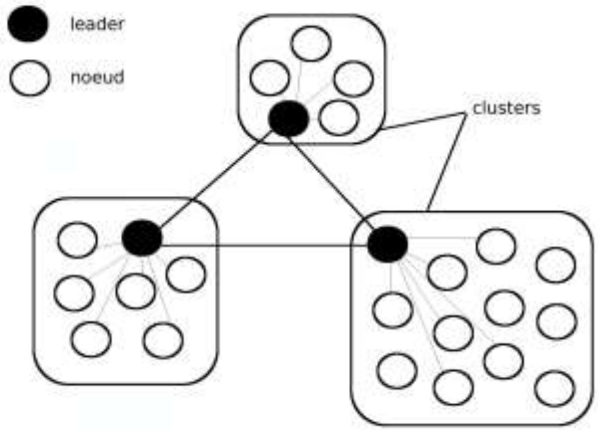

Fig 2: Hierarchical checkpointing for grids

IV. Performance evaluation

In the most previous studies, fault tolerance algorithms were tested in flat architectures, namely in a cluster. The aim of our study is to determine which algorithm best suits the architectural grid. To this aim, we implement the seven checkpoint algorithms described in Section 2: the 3 main messages logging protocols (represented as “ML” in the figures), Chandy-Lamport, Communication induced protocol (CIC in figures), and blocking coordinated checkpointing.We compare the performance of these algorithms in cluster and grid environments. We use the Omnet++ simulator [7]. The cluster is configured with 25 nodes. For the grid configuration, 25 nodes were uniformly spread in 5 clusters. The intra-cluster delay is fixed to 0.1 ms and the inter-cluster delay is fixed to 100ms. Our tests were carried out with 50 application processes. Messages between processes were randomly generated.

A. *Failure free performance*

Fig. 3 presents the performance of the algorithms in both configurations. It is obvious that the time taken to run an application with checkpointing is greater than the time taken for it to run without checkpoint. Protocol overhead checkpoint coordinated non-blocking is less compared to other approaches to that phase synchronization is limited to the cluster and the second concerns only the leaders of each cluster. The additional cost of communications-driven approach is due to the forced checkpoints during execution. Logging protocols are sensitive to characteristics of the application, especially in communications-intensive applications. Indeed, they produce a large overhead due to the backup of messages on stable storage and the increasing size of messages to piggyback determinants.

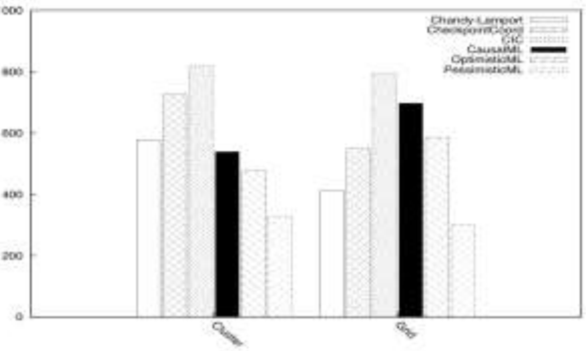

Fig. 3: Failure free performance, Checkpoint interval=180s, Execution time=900s

B. Recovery time

The recovery time depends on the number of checkpoints maintained by the protocol and the number of rollbacks. In coordinated checkpointing and pessimistic logging, recovery is simplified because the system is rolled back only to the last recent checkpoint. In the grid approach, the additional cost of recovery decreases slightly. Indeed, if the faulty node has no dependencies with nodes of other cluster nodes, the fault is confined to the cluster node's fault. So all the nodes of the grid do not perform the recovery procedure. By cons, if the inter-cluster communications are intensive, the overhead increases as in the case of causal and optimistic logging.

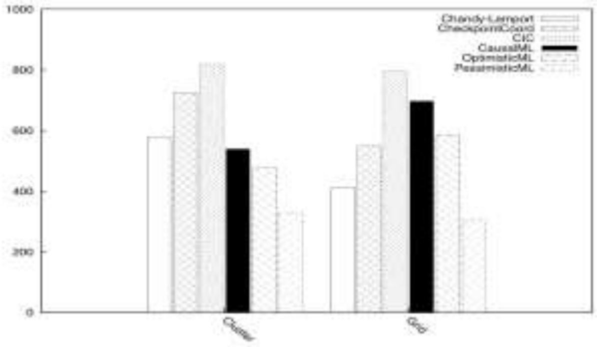

Fig.4: Overhead of recovery,checkpoint interval=180s,execution time=900s,numbers of fault=10

C. Number of rollbacks

For coordinated checkpoint protocols, all processes must resume during recovery. The logging protocol reduces the number of rollback. This number is minimal in pessimistic approach since only faulty processes need to be rolled back. For the other logging protocol, this number depends on the information stored in backups and in the main memory of correct processes.

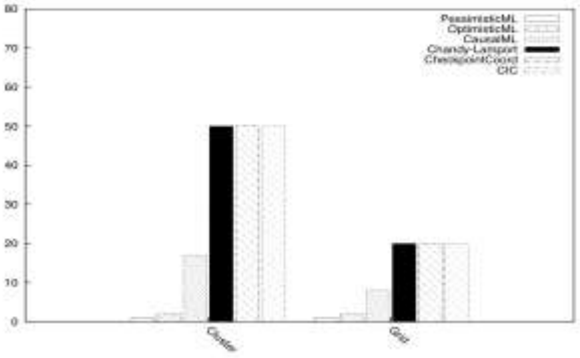

Fig. 5: Number of process, Checkpoint interval=180s, Execution time=900s, Numbers of fault= 1

V. RELATED WORK

Paul et a.l [4] proposes a hierarchical protocol based on coordinated checkpoint. This protocol is designed for hierarchical networks like the Internet. The experiments were made on a network of four clusters of eight nodes. Authors consider three roles of the different processes. *Initiator* is the process that initiates checkpoint sessions. One *Leader* process coordinates the activities within each cluster, in line with the instructions of the *Initiator*. *Follower* are the rest of the system processes, they follow the instructions of their *Leader*. The checkpoint protocol is hierarchical in two phases. The first phase is the execution of the algorithm coordinated checkpoint limited to the cluster. During this phase the processes are blocked and establish a consistent checkpoint. The second phase is a coordinated checkpoint but the leaders are the only participants, with the initiator, which acts as a coordinator. The experiments showed that the overhead of checkpointing in the hierarchical approach is lower than in the standard “flat” coordinated protocol. However the protocol hierarchy is sensitive to the frequency of messages between clusters. Indeed the extra cost of checkpoint increases progressively as the frequency of messages increases, and tends towards that of the checkpoint protocol standard.

Bhatia et al. [5] propose a hierarchical causal logging protocol that addresses the scalability problems of causal logging. Indeed, the traditional causal logging algorithms are used successfully in small-scale systems. They are known to provide a low overhead during failure-free executions sending no extra messages. But they are not scalable since each application process needs to maintain a data structure, which grows quadratically with the number of processes in the system.

Authors reduce the data structure by an exponential amount. They propose a hierarchical approach using a set of proxies spread on the network that act as a distributed cache. This approach highly reduces the amount of information piggybacked on each messages. However, the use of proxies decreases the performance of recovery since the recovery information is spread on the proxies.

Monnet et al. [6] propose a hierarchical checkpointing protocol, which combines coordinated checkpointing inside clusters and a checkpoint induced by communications between clusters. Simulation of the protocol shows that it generates a high number of forced checkpoints when the communication rate between clusters increases. Then, this approach is more suitable for code coupling applications where communications are mainly local inside clusters.

Several techniques are used to implement fault tol- erance in message-passing systems. Simple replication is not relevant for such systems, since if the system is designed to tolerate *n* faults, every component must be replicated *n* times and the computation resources are thus divided by *n*. The two main techniques used are message- logging and coordinated checkpoints. A review of the different techniques can be fount in [2].

Message-logging consists in saving the messages sent between the computation nodes, and replay them if a failure occurs. It is based on the *piecewise deterministic assumption*: the execution of a process is a sequence of deterministic events separated by non deterministic ones [14]. With this assumption, replaying the same sequence of non-deterministic events at the same moment makes possible the recovering of the state preceding a failure. Thus these protocols consist for every process to save

all its non-deterministic events in a reliable manner and to checkpoint regularly. When a failure occurs, only the crashed process is restarted from its last checkpoint, and it recovers its last state after having replayed all saved events. There is no need to coordinate the checkpoints of the different processes. No orphan processes (*i.e.* processes that are waiting for a message that will never come, since the expected sender is more advanced into its execution) are created. The recover mechanism is more complex than with coordinated checkpoints as a process shall obtain its past events and be able to replay them. Moreover the overhead induced during failure-free execution decreases the performances in not very faulty environments, such as clusters [23]. Furthermore, it can lead to the domino effect [24]: a process that rollbacks and that need a message to be replayed, asks another process to rollback. This process does, and asks another one to do so, etc. The execution can be restarted from the beginning because of cascading rollbacks and so the benefits of fault tolerance are lost.

Message-logging protocols are classified into three categories : optimistic, pessimistic and causal proto- cols. Optimistic protocols assume that no failure will occur between the moment a process executes a non- deterministic event and the moment this event is saved on a reliable storage support. So when a process executes a non deterministic event, it sends it to the reliable storage support then continues its computation without waiting any acknowledgment [22]. The induced overhead during failure-free execution is then quite small, but the optimistic hypothesis introduces the risk to get an incoherent state if it is not realized. Pessimistic protocols do not make this hypothesis, and the processes wait for an acknowledgment from the reliable storage support to continue their execution [23]. The induced overhead during fault-free execution is then important. The third category of message-logging protocols tries to gather the advantages of both optimistic and pessimistic protocols: low overhead during fault-free execution, and no risk to recover into an incoherent state. It consists in saving the causality information on a reliable storage, but does not need to wait for the acknowledgment from this medium by piggybacking these information in the messages until the acknowledgments are received. A description can be found in [24], and another causal protocol based on dependencies graphs is described in [25]. A metric to evaluate the performances of message-logging protocols can be found in [26].

Coordinated checkpointing has been introduced by Chandy and Lamport in [27]. This technique requiresthat at least one process sends a marker to notify the other ones to take a snapshot of their local state and then form a global checkpoint. The global state obtained from a coordinated checkpoint is coherent, allowing the system to recover from the last full completed checkpoint wave. It does not generate any orphan processes nor domino effect, but all the computation nodes must rollback to a previous state. The recover process is simple, and a simple garbage collection reduces the size needed to store the checkpoints.

In blocking checkpointing protocols, the processes stop their execution to perform the checkpoint, save it on a reliable storage support (that can be distant), send an acknowledgment to the checkpoint initiator and wait for its commit. They continue the execution only when they have received this commit. The initiator sends the commit only when it has received all the acknowledg- ments from all the computing nodes to make sure that the global state that has been saved is fully completed. As claimed in [28], blocking checkpoints induce an important latency and non-blocking checkpoints are then more efficient.

Non-blocking coordinated checkpoints with dis- tributed snapshots consists in taking checkpoints when a marker is received. This marker can be received from a centralized entity, that initiates the checkpoint wave, or from another computation node which has itself received the maker and transmits the checkpoint signal to the other nodes. This algorithm assumes that all the communication channels comply with the FIFO property. Therefore the computation processes do not have to wait for the other ones to finish their checkpoint, and then the delay induced by the checkpoint corresponds only to the local checkpointing.

Communication-induced checkpoint protocols (CIC) perform uncoordinated checkpoints but avoid the domino effect [29]. Unlike coordinated checkpoints, it does not require additional messages for a process to know when it has to perform a local checkpoint. The information about when a local checkpoint must be performed are piggybacked in the messages exchanged between the processes. Two kinds of checkpoints are defined: local and forced. Local checkpoints are decided by the local process, forced ones are decided by the process accord- ing to the information piggybacked in the messages. The forced ones avoid the domino effect and ensure then the progress of the recovery line, i.e. the set of checkpoints of all the processes describing a coherent global state. When a failures occurs, all the processes rollback to their last stored local checkpoint and then to the last recovery

line. CIC is an interesting theoretical solution but it has been shown in [30], using NPB 2.3 benchmark suite [31], that it is not relevant for typical cluster applications.

Several MPI libraries are fault tolerant. A review can be found in [32]. Coordinated checkpointing has been implemented in several MPI implementations on different levels of the application.

LAM/MPI [33], [34] implements the Chandy-Lamport algorithm for a system-initiated global checkpointing. When a checkpoint must be performed, the *mpirun* process receives a checkpoint request from a user or from the batch scheduler. It propagates the checkpoint request to each MPI process to initiate a checkpoint wave. As in our blocking Chandy-Lamport implementation, each MPI process then coordinates itself with all the others, flushing every communication channel, in order to reach a consistent global state. If a failure occurs, mpirun restarts all the processes from their last stored state. Finally, processes rebuild their communication channels with the other ones and resume their execution.

VI. Conclusion

In this paper, we compare checkpoint protocols and message logging in grid computing. We propose a hierarchical approach to combine different algorithms. We find that the protocols that require the recovery of all processes in case of single failure are poorly suited to systems with many processes. The message logging protocols are more suitable for large configuration with the exception of some causal logging approach, which induces communications to all processes during the recovery. Non-blocking coordinated checkpoint are not sensitive to the rate of communications. They therefore represent an attractive solution for applications and highly interconnected grid architectures by reducing the number of markers sent during the synchronization phase.

References

[1] E. N. (MOOTAZ) ELNOZAHY, LORENZO ALVISI, YI-MIN WANG, DAVID B. JOHNSON, A Survey of Rollback-Recovery Protocols in Message-Passing Systems, ACM Computing Surveys, Vol. 34, No. 3, September 2002, pp. 375–408.

[2] Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Syst. 31, 1, 63–75.

[3] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, page 127, New York, NY, USA, 2006. ACM.

[4] Himadri S. Paul, Arobinda Gupta R. Badrinath, Hierarchical Coordinated Checkpointing Protocol. In International Conference on Parallel and Distributed Computing Systems, pages 240-245, November 2002

[5] K. Bhatia, K. Marzullo, and L. Alvisi. “Scalable causal Message Logging for Wide-Area Environments,” Concurency and Computation: Practice and Experience, 15(3), pp. 873-889, Aug. 2003.

[6] S. Monnet, C. Morin, R. Badrinath, “Hybrid Checkpointing for Parallel Applications in cluster Federations”, Proc. 4th IEEE/ACM International Symposium on Cluster Computing and the Grid, Chicago, IL, USA, pp. 773-782, April 2004.

[7] http://www.omnetpp.org

[8] Open Grid Forum. http://ogf.org

[9] Organization for the Advancement of Structured Information Standards (OASIS). http://www.oasis-open.org

[10] Internet Engineering Task Force. http://www.ietf.org

[11] A. Avizienis, J. Laprie, B. Randell and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing”, IEEE Transactions on Dependable and Secure Computing, 1(1), pp. 11-33, 2004

[12] A. Borg, J. Baumbach, and S. Glazer, “A message system supporting fault-tolerance”, In Proceedings of the Symposium on Operating Systems Principles, ACM SIGOPS, pp. 90-99, Oct. 1983

[13] M. L. Powell, and D. L. Presotto, “Publishing: A reliable broadcast communication mechanism”, In Proceedings of the Ninth Symposium on Operating System Principle, ACM SIGOPS, pp. 100-109, Oct. 1983

[14] R. B. Strom and S. Yemeni, “Optimistic recovery in distributed system” ACM Transactions on Computer Systems, 3(3), pp. 204-226, April 1985

[15] D. B. Johnson and W. Zwaenepoel, “Sender-based message logging”, In Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing, IEEE Computer Society, pp. 14-19, June 1987

[16] R. E. Strom, D. F. Bacon and S. A. Yemeni, “Volatile logging in n-fault-tolerant distributed systems”, In Proceedings of the Eighteenth Annual International Symposium on Fault-Tolerant Computing, pp. 44-49, 1988

[17] A. P. Sistla and J. L. Welch, “Efficient distributed recovery using message logging”, In Proceedings of the Eighth Symposium on Principles of Distributed Computing, ACM SIGACT/SIGOPS, pp. 223-238, Aug. 1989

[18] D. B. Johnson, and W. Zwaenepoel, “Recovery in distributed systems using optimistic message logging and checkpointing”, Journal of Algorithm, 11: pp. 462-491, 1990

[19] S. Venkatesan, and T. Y. Juang, “Efficient algorithms for optimistic crash recovery”, Distributed Computing, 8(2): pp. 105-114, June 1994

[20] E. N. Elnozahy and W. Zwaenepoel, “In the use and implementation of message logging”, In Digest of Papers: 24 Annual International Symposium on Fault-Tolerant Computing, IEEE Computer Society, pp. 298-307, June 1994

[21] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, “Improved message logging versus improved coordinated checkpointing for fault tolerant MPI,” in *IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE CS Press, 2004

[22] B. Randell, “System structure for software fault tolerance,” *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 220–232, 1975

[23] A . Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, “MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based mes- sage logging,” in *High Performance Networking and Computing (SC2003)*, Phoenix USA. IEEE/ACM, November 2003

[24] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello, “Coordinated checkpoint versus message log for fault tolerant MPI,” in *IEEE International Conference on Cluster Computing (Cluster 2003)*. IEEE CS Press, December 2003

[25] E. N. Elnozahy and W. Zwaenepoel, “Replicated distributed processes in manetho,” in *22nd International Symposium on Fault Tolerant Computing (FTCS-22)*. Boston, Massachusetts: IEEE Computer Society Press, 1992, pp. 18–27

[26] L. Alvisi and K. Marzullo, “Message logging: Pessimistic, optimistic, causal, and optimal,” *IEEE Trans. Software Eng*, vol. 24, no. 2, pp. 149–159, 1998

[27] K. M. Chandy and L.Lamport, “Distributed snapshots : Determining global states of distributed systems,” in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75

[28] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel, “The performance of consistent checkpointing,” in *Symposium on Reliable Distributed Systems*, 1992, pp. 39–47

[29] J.-M. Helary, A. Mostefaoui, and M. Raynal, “Communication- induced determination of consistent snapshots,” *IEEE Transac- tions on Parallel and Distributed Systems*, vol. 10, no. 9, pp. 865–877, 1999

[30] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. D. Mel, “An analysis of communication induced checkpointing,” in *29th Symposium on Fault-Tolerant Computing (FTCS’99)*. IEEE CS Press, june 1999

[31] N. A. R. center, “Nas parallel benchmarks,” 1997http://science.nas.nasa.gov/Software/NPB/

[32] W. Gropp, and E. Lusk, “Fault tolerance in MPI Program”, Special issue of the Journal of High Performance Computing Applications (IJHPCA), 2002

[33] G. Burns, R. Daoud, and J. Vaigl, “LAM: An Open Cluster Environment for MPI,” in *Proceedings of Supercomputing Symposium*, 1994, pp. 379–386

[34] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, “The LAM/MPI checkpoint/restart framework: System-initiated checkpointing,” in *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003

[35] S. Zanikolas, and R. Sakellariou, “A taxonomy of Grid monitoring systems”, Future Generation Computer System, 21(1), pp. 163-188, 2005

[36] H. Jitsumoto, T. Endo, and S. Matsuoka, “ABARIS : An adaptable fault detection/recovery component framework for MPI”, Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IEEE Computer Society Press : Los Alamitos, CA, pp. 1_8, 2007

[37] H. Jin, W. Qiang, and D. Xou, “DRIC : Dependable Grid Computing framework”, IEICE Transactions on Information and System : E89-D(2), pp. 612-623, 2006

[38] E. Elnozahy, D. Johnson, and Y. Wang, “A survey of rollback recovery protocols in message passing systems”, ACM Computing Surveys, 34(3), pp. 375-408, 2002

[39] I. Foster and C. Kesselman. The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, ISBN 1558609334, 2003

Ndeye Massata Ndiaye received his B.Sc and M. Sc in Cumputer science from Gaston Berger University of Saint and M.Phil in Computer Science from Cheikh Anta Diop university of Dakar in Senegal. She is now an assistant professor in university of Bambey Senegal.

Pierre Sens is a full professor at the University of Paris 6 since 2003. He received his PhD in 1994 and the "Habilitation à Diriger des Recherches" in 2000. Since 2002, he leads Regal project which is a joint research team between LIP6 and INRIA, France. He has been author and co-author of published papers in several books, journals and recognized international conferences and symposiums.

Ousmane Thiare. Received a PhD in computer science (Distributed systems) at 2007 from the university of Cergy Pontoise, France. He is an associate professor in Gaston Berger University of Saint-Louis Senegal. He has been co-author of published papers in several journals and recognized international conferences and symposiums.