# Enhanced Coordinated Checkpointing in Distributed System

Bakhta Meroufel and Ghalem Belalem

***Abstract*—**Coordinated checkpointing is a well-known method for achieving fault tolerance in distributed computing systems. This type of checkpointing selects an initiator to manage and ensure the checkpointing process. The majority of existing works ignore the role and the importance of this initiator. The work presented in this paper can be divided on two parts. In the first part, we examine the impact of initiator choice on different types of coordinated checkpointing and we prove its importance in term of performances. We propose also a simple and an effective strategy to select the best initiator each checkpointing round. In the second part of this work, we focused on the soft checkpointing and we have strengthened the role of initiator by adding a storage manager that ensures atomicity and speed of storage checkpoints files using a smart I/O strategy.

***Keywords*—**Checkpointing, consistency, rollback, fault tolerance, overhead, initiator, coordination, I/O, atomicity, data sieving, collective I/O.

## I. Introduction

Since the computing potential of distributed systems is often hindered by their susceptibility to failures, many different techniques of fault tolerance have been developed and integrated into them accordingly, in order to improve both their reliability and availability and to reduce re-computations. Fault-tolerance is the property that enables a system to continue operating properly in the event of the failure of some of its components. To achieve the fault tolerance, several techniques are proposed such as: replication and checkpointing [14].

Replication or redundancy creates many identical and consistent copies of the object (data or task) in several resources, in case of failure, one copy can replace the failed one. The replication ensures a real-time fault tolerance by masking the failure but it consumes additional resource and increases the system complexity. The second technique is the checkpointing. Checkpoint/Restart (C/R) is a way to provide persistence and fault tolerance in both uni-processor and distributed systems. Checkpointing is the act of saving an application's state to stable storage during its execution, while restart is the act of restarting the application from a checkpointed state. If checkpoints are taken, then when an application fails, it may be possible to restart it from its most recent consistent checkpoint. This limits the amount of computation lost because of a failure to the computation performed between the last checkpoint and the failure.

In a distributed system, since the processes in the system do not share memory, an $i^{th}$ global state $GS_i$ of the system is defined as a set of local states $LS_j$, one from each process j participating in the application:

$$GS_i = \{LS_1, LS_2, \ldots, LS_n\} \quad (1)$$

A checkpointing must create a consistent state. In this case, GSi must be created without any orphan message. GSi is strongly consistent if it is consistent and without any transit messages. $Send_a(m)$ and $Recv_b(m)$ note the send and the receive events respectively in the system. So:

The message m is orphan if:

$$Recv_b(m) \in LS_b \text{ and } Send_a(m) \notin LS_a \text{ and } \{LS_a, LS_b\} \subseteq GS_i \quad (2)$$

The message m is transit if:

$$Recv_b(m) \notin LS_b \text{ and } Send_a(m) \in LS_a \text{ and } \{LS_a, LS_b\} \subseteq GS_i \quad (3)$$

To satisfy that consistency, the checkpointing technique can be classified in three categories:

- Independent checkpointing (Uncoordinated) [4][5]: the checkpoints at each process are taken independently without any synchronization among the processes. Because of absence of synchronization, there is no guarantee that a set of local checkpoints taken will be a consistent set of checkpoints. It may require cascaded rollbacks that may lead to the initial state due to domino-effect [4]. The domino effect appears when a subset of processes rollback unboundedly to determine a set of mutually consistent checkpoints. The independent checkpointing store all the checkpoints file during the job life.
- Communication induced checkpointing: the processes take two kinds of checkpoints, local and forced. Local checkpoints can be taken independently, while forced checkpoints are taken to guarantee the eventual progress of the recovery line. However the messages are piggybacked and useless checkpoints can be created.

**B. Meroufel**, Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran 1 – Ahmed Ben Bella, Oran, Algeria BP.1524, EL M'Naouer, 31000, Oran, Algeria. (email: bakhtasba@gmail.com).

**G. Belalem**, Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran 1 – Ahmed Ben Bella, Oran, Algeria BP.1524, EL M'Naouer, 31000, Oran, Algeria. (email: ghalem1dz@gmail.com).

- Coordinated or synchronous checkpointing [1][12]: the processes will synchronize to take checkpoints in such a manner that the resulting global state is consistent. The main advantage is that it sores only one permanent checkpoint in the stable memory and it is domino-effect free.

The experimental results in the literature prove that the coordinated checkpointing is the most convenient checkpointing strategy in scalable distributed system. It minimizes the resource consumption and ensures a consistent state using only one recovery line. Checkpointing usually consists of three main phases: consistency management phase, checkpointing creation phase and storage phase. The major problem of checkpointing is the overhead caused by the storage time of checkpointing files in stable storage (70% of checkpointing time is caused by the storage [16]). During the storage time, and whatever the checkpointing protocol used (coordinated/ uncoordinated/ communication indexed), nodes will be blocked to ensure the checkpointing atomicity, which significantly increases application execution time and causes overhead.

To reduce the time of checkpointing storage, several strategies are proposed in the literature. The soft checkpointing is a powerful strategy to reduce the storage time. Generally, in conventional checkpointing, each node involved in the checkpointing, creates and stores its own checkpointing file in stable storage server. In the soft checkpointing, nodes create their files and send them to a special node, and then they resume their works immediately. The special node collects the files and stores them in the stable memory parallel with the execution of application [18]. The management of I/O is another strategy that allows the reduction of the storage time of checkpoints by reducing the transfer time and the data quantity needed for the storage [17].

In this paper we will focalize on the coordinated checkpointing. Our contribution is divided on two parts: the initiator choice and the soft checkpointing improvement. In the initiator choice, we will study the different strategies of this type of checkpointing, not to compare between them, but to study the impact of initiator choice on the performances of each strategy. We will propose an approach to improve the checkpointing and the recovery performances. In the part of soft checkpointing improvement, we will strengthen the role of initiator by adding a storage manager. The storage manager ensures the atomicity and uses a smart I/O to manage the storage phase.

The remainder of the paper is organized as follows. Section 2 reviews some coordinated checkpointing strategies existing in literature and explains also several I/O techniques. Section 3 describes in details the most popular coordinated checkpointing. Section 4 introduces our approach of initiator selection. Section 5 explains the soft checkpointing based on our initiator. Section 6 shows some experimental results and analysis the performances of our both contributions. Finally, a conclusion and some perspectives are given in section 7.

## II. Related Works

The coordinated checkpointing ensures the consistency using two types of coordination: blocking and non blocking. A blocking, coordinated checkpointing protocol requires flushing communication channels before taking the state of a process in order to ensure the channels during the checkpoint without interrupting the computation. It requires logging in-transit messages and replaying them at restart, which implies coordination with the progress engine and queue mechanisms. The Chandy-Lamport [2] algorithm is the earliest non-blocking all-process coordinated checkpointing algorithm. This technique requires that at least one process sends a marker to notify the other ones to take a snapshot of their local states and then forms a global checkpoint. Since markers are sent along all channels in the network, this algorithm leads to an extra-overhead [1].

To minimize the checkpointing overhead for both blocking and non-blocking algorithm, it was necessary to reduce the number of nodes involved in the checkpointing process. In this case, the system must track all the dependencies created during a checkpointing interval [8]. Only the nodes depending directly or transitively to a chosen initiator will be forced to create their checkpoints. Koo and Toeg [6], and Cao and Singhal [7] proposed minimum-process blocking coordinated checkpointing algorithms. In [9], [10], the authors proposed a non-coordinated checkpointing with the minimum process in mobile distributed systems. The min-process checkpointing cause the creation of useless checkpoints. To overcome this problem, Cao and Singhal [11] introduce the concept of mutable checkpoints. The authors in [12] use a probabilistic approach to control the creation of mutable checkpoints. A process takes its mutable checkpoint only if the probability that it will get the checkpoint request in the current initiation is high. In [13], the authors combine between min- process and all-process checkpointing to reduce the rollback overhead. The time based coordinated checkpointing [13] suppose that the nodes among the system have not have loosely synchronized clocks. In this case, a simple control of send/receive events during a known period can create a consistent state.

The majority of existing coordinated checkpointing use two-phases checkpointing to reduce the overhead. The first phase creates tentative checkpoints TCP. It is a checkpointing file stored only in the local memory of the node. It will be transformed to permanent checkpoints PCP in the second phase by storing the TCP in the stable memory.

Ensure consistency and creating checkpoints presents only 30% of the time checkpoints, the remaining time is caused by the checkpoints storage phase. Minimizing the checkpoint file size can reduce the storage time. The compression of the checkpoint files and reducing the number of nodes involved in the checkpoints can minimize the checkpoint file size.

Another way to reduce the overhead is to store only fundamental data at each checkpoint. The ability to identify these data depends on the checkpoint level Incremental checkpointing [19] reduces this cost. A runtime monitor tracks application writes, and if it detects that a given memory region has not been modified between two adjacent checkpoints, that region is omitted from the subsequent checkpoint, thereby reducing the amount of data to be saved. This strategy can be expensive since it tracks all the user's operations.
The management of I/O has proven its efficiency in term of storage time reduction. The I/O can be used in the general cases where there are write/read executions (even without checkpointing) and is generally based on the notion of aggregation. The I/O management can be also hybridized with the other strategies such as the compression and even the incremental checkpointing which makes it very useful and powerful strategy.

An important reason for the limitations of classical I/O systems is that applications often send smaller queries disjoint. This access mode generates a first additional cost to the large number of applications running on various transmission channels (bus / network communications), but more significantly increases the processing time of the latter [14]. To deal with this problem, several "aggregation" methods have been proposed we can distinguish two types of aggregations strategies: dependent and collective.
Independent I/O is a straightforward form of I/O and is widely used in parallel applications. This form of I/O can be called independently by an individual process or any subset of processes of a parallel application. The advantage of independent I/O is that users have the freedom to perform I/O for each individual process or any subset of the processes that open the file. The buffering is an Independent I/O [24]. In conventional strategies, the write operation transfers data from the buffer to the local disk from their reception. Buffering proposes that the buffer will be used for temporary storage of I/O. The write operation includes small blocks in a buffer (of limited size). Once the buffer is completely filled, it will be forwarded to the local disk.
The "List I/O" approach [23] provides routines to indicate within a single call access number. A list of coherence of the view built [5]. It introduces synchronization in the distributed system while communications are frozen. However, since it does not require copies of incoming or outgoing messages, it is simpler to implement in an existing high-performance communication driver [3][4][5].
A non-blocking, coordinated checkpointing protocol consists of saving the state of the communication torque (offset, size) describes the distribution of data in memory and a similar list is used to perform matching on disk. In this strategy, the messages will piggyback a lot of data during the I/O which increases the overhead.
Data sieving [22] is one of the techniques proposed to address this issue by aggregating small requests into large ones. Instead of accessing each small piece of data separately, data sieving accesses a large contiguous scope of data that includes the small pieces of data. The additional unrequested data are called holes (See Figure 1). The size of holes compared to the requested data controls the efficiency of data sieving.

For many parallel applications, even though each process may access several non-contiguous portions of a file, the requests of multiple processes are often interleaved and may constitute a large contiguous portion of a file together [20]. In order to achieve better I/O performance, a group of processes may cooperate with each other in reading or writing data in a collective and efficient way, which is known as collective I/O.

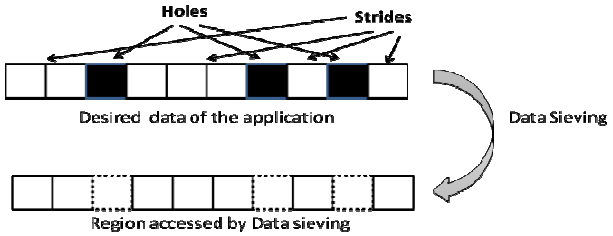


*Fig. 1: Data sieving approach [15]*

The collective I/O is a general idea that exploits the correlations among accesses from multiple processes of a parallel application and optimizes its I/O accesses. The basic idea behind this technique is to coordinate I/O accesses from different processors. The processors exchange information regarding what data each of them needs to access. This information is used to derive an efficient I/O schedule. Note that an I/O schedule may require a processor (aggregator) to access data on behalf of some other processors which results in communication when executing the I/O schedule.
The collective I/O can distinguished by the "physical place" where the operation group is performed [20]: if aggregation is executed among processes (calculation on the nodes), the most used method is the "Two-Phase I/O" approach; if aggregation is performed at the records, we are talking about system "Disk-Directed I/O" if the approach is finally realized within a server, the method is "server-directed I/O".
"Two-Phase I/O", this method [21], as its name suggests, consists of two main phases: after a consensus between the processes involved, the first step is to retrieve the data, the second concerns the redistribution between each of the latter processes. To implement the first phase, each process must know the necessary data to others. The advantage of this method is that it allows to access contiguous and wide and therefore strongly reduce the time of reactivity. This method is the most portable strategies which makes it desirable for the heterogeneous systems (See Figure 2).

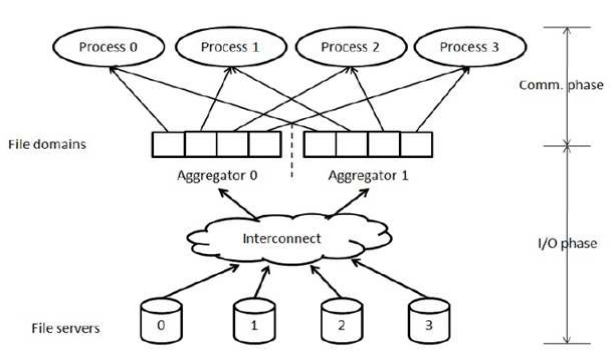


*Fig. 2: Collective I/O*

ROMIO is the most popular I/O platform used in distributed system [22]. It uses the collective I/O (Two-phases) to manage the requested blocks of different nodes and then it uses the data sieving to transfer these blocks to the memory.

## III. Coordinated Checkpointing

A message passing system consists of N fixed number of nodes that communicate each other only through messages. The messages generated by underlying distributed application will be referred to as computation messages. Messages generated by the nodes to advance checkpoints, handle failures and for recovery will be referred to as system messages. In this paper the horizontal lines extending towards right hand side represent the execution of each process and arrows between them represent the messages. Processes have access to a stable storage device that survives failures. The Figure 3 illustrates the communication process of three nodes. The message *m1* is in-transit message, and *m3* is an orphan message according to the recovery C1. The messages *m0* and *m2* are regular messages.

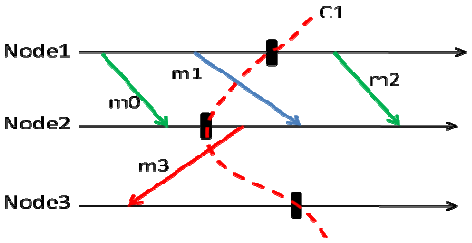


*Fig. 3: System presentation*

In the next section, we will explain the two most popular and performing coordinated checkpointing types: blocking and non-blocking.

### A. BlockingCoordinated Checkpointing

In case of Minimum blocking coordinated checkpointing and after selecting initiator (See Figure 4, step-1), the initiator sends "Request" to the dependent nodes (See Figure 4, step-2). Dependencies are identified by the Dependency matrix. The dependency matrix of Figure 4 is presented in formula (4).

$Dep_{Matrix}[i][j] = 1$ means that the node i depends on node j (the node j sent a message to node i during the current checkpointing interval).

$$Dep_{Matrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

When a node receives this message, it freezes its communication and creates its TCP then it returns "Response" to the initiator (Figure 4, step-3).

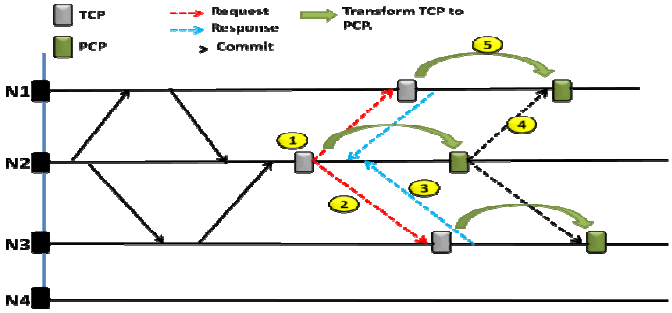


*Fig. 4: Blocking coordinated checkpointing*

The word freeze indicates that the node blocks its execution when it finds a communication event (sends / receives). The initiator collects all the "Response" and sends "Commit" to nodes (See Figure4, step-4). This message informs the receiver to transform its TCP to PCP and then continues its running task (See Figure, step-5).

In case of All blocking coordinated checkpointing, the algorithm is the same except that all nodes are involved in the checkpointing (See Formula 5):

$\forall(i,j) \in n, Dep_{Matrix}[i][j] = 1$ (5)

Where n is the number of the nodes that run the application.

### B. Non-Blocking Coordinated Checkpointing

After the selection of initiator, the initiator sends "Request" to the dependent nodes (See Figure 5, step-1) using the dependency matrix in formula (6). Since checkpointing is not blocking, nodes can communicate with each other during the checkpointing process.

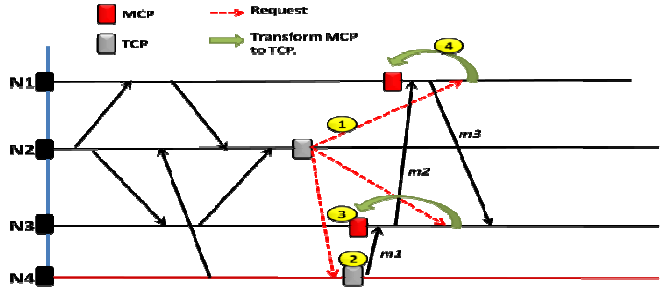


*Fig. 5: Non-Blocking coordinated checkpointing*

To ensure the creation of a coherent state, the checkpointing uses "piggybacked messages." Among piggybacked information there is the *csn* (checkpointing sequence number). The Integer $csn_i$ keeps track of sequence number of the current checkpoint of process $N_i$. It is initialized to 0 and increased by one each time a new tentative checkpoint is taken.
If a node receives a message with higher *csn* compared to its local *csn*, he knows that the sender of this message has already created his checkpoint before sending this message. In the classical coordinate checkpointing approach and in this situation, the receiving node is forced to create his checkpoint and increment its *csn* to prevent the creation of orphan messages. In this case, several unnecessary checkpoints can be created because it is possible that the sender is not concerned by the checkpointing. For this reason, it is preferable to use the Mutable checkpoint MCP, which is neither a tentative checkpoint nor a permanent checkpoint. Mutable checkpoints can be saved anywhere, e.g., the main memory or local disk and the effort of creating it (mutable checkpoint) is negligible as compared to the tentative one [11]. Figure 6 illustrates the three checkpoint types that can be created according to the message type. Some criteria must be satisfied before creating this MCP such as: the csn of the received message is higher than the local csn; the node has send a message to other nodes during the last checkpointing interval and the current checkpointing process has not finished yet [11].

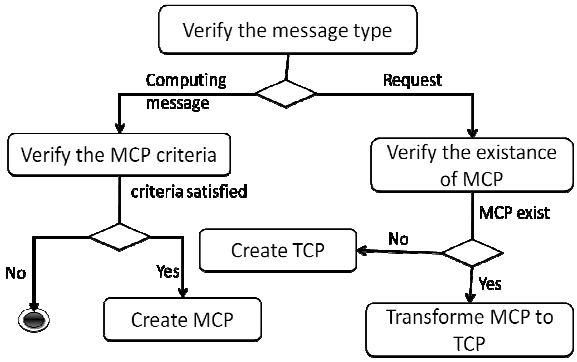


*Fig. 6: Mutable checkpointing technique*

In the case of node *N4* in Figure 5, he received the "Request", creates its TCP (See Figure 5, step-2), increments its *csn* (*csn = 1*) and then sends m1 to *N3*. Node *N3* receives m1 before the request checkpointing. In this case it only creates an MCP (See Figure 5, step-3; same for the node *N1*). The MCP will be converted to TCP when the node receives a "Request" checkpointing (See Figure 5, step-4). If after a timeout, the request is not received, MTC will be deleted without any overload. In case of distributed checkpointing, nodes that create MTC do not broadcast the checkpointing requests [15].

$$Dep_{Matrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (6)$$

After the creation of TCP, the node sends "Response" to the initiator (this phase is not shown in Figure 5). The initiator collects the responses and sends "Commit" to the concerned nodes to transform their TCP to PCP.
In case of all non blocking coordinated checkpointing, the algorithm is the same except that all nodes are involved in the checkpointing (See Formula 5).

## IV. Contribution1: Initiator Choice

In Coordination checkpointing strategies, the initiator has an important role in the process since it:

- Starts and controls the checkpointing process
- Determines the involved nodes in the checkpointing round
- Ensures the checkpointing atomicity.
- Ensures the checkpointing storage in case of soft checkpointing
- Declares the checkpointing termination.
- Manages the inter-group checkpointing and ensures the intra-group checkpointing in case of hierarchical system.
- Its clock is used generally as reference for resynchronization phase in case of time based coordinated checkpointing.

Despite the important role of initiator in the coordinated checkpointing, the majority of existing studies do not take into account the strategy of initiator selection. When all nodes decide to create their checkpoints, the system or the checkpointing manager selects the initiator either randomly or based on the smallest identifier among the candidates. During our research, we find a single paper that proposed a strategy for the selection of initiator.
The paper [15] uses the same strategy described in Section 3.2 with one difference: the selection of initiator is based on popularity. Each node $N_i$ uses a Boolean dependency vector $Vect_i$ of size n where n is the number of nodes in the system. This vector is initialized as follows (See Formula 7):

$$\begin{cases} Vect_i[j] = 1 \;\; if \; i = j \\ \quad else \\ Vect_i[j] = 0 \end{cases} \quad (7)$$

At the reception of a computing message sent by $N_j$ to $N_i$, the $j^{th}$ column of $Vect_i$ will be equal to 1. $Vect_i$ will be initialized after each checkpointing and it is piggybacked in every computing message. The number of 1 in the vector indicates the popularity of the node (See Figure 7). The grouping of dependency vectors built dependency matrix.

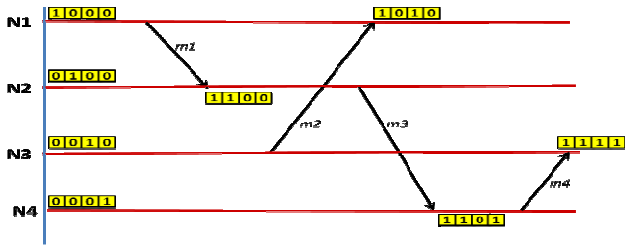


*Fig. 7: Dependency vector*

According to the algorithm proposed in [15], if the node exceeds a certain threshold of popularity (in paper experiments threshold =*50%*), it triggers the checkpointing. The experimental results of this paper have shown that the choice of initiator has a great impact on checkpointing. However, this strategy may cause a conflict where a many initiators may exist at the same time. This requires a centralized decision to eliminate this problem. In addition, this strategy can not decrease the gap between checkpoint sequence numbers.

To overcome these problems, we propose our strategy of initiator selection, where the initiator is the node with a small csn and a great speed (MIPS) as presented in Formula 8.

$$Init^k = \max_{i \in N}\left(\frac{S_i}{csn_i}\right) \quad (8)$$

Where:

- k: id of checkpointing round; it also represents the new csn. We suppose that csn in initialized at 1.
- N: set of nodes in the application.
- $S_i$: speed of node i in MIPS
- $csn_i$: csn of node i.

Our approach uses $csn_i$ as criteria for initiator choice, thereby avoiding the problem of famine where only a few nodes create their checkpoints each round. In the case of the strategy proposed in [15], there will be nodes that rarely create their checkpoints because they often do not communicate with the initiator (directly or transitively). In case of rollback, the rate of re-computing for these nodes will be high which increases the total checkpointing overhead. Selecting initiator based on speed accelerates the checkpointing because it is its role to treat all data of involved nodes during the checkpointing process.

## V. Contribution2: Soft Checkpointing Improvement

In the previous section, we used the hard checkpointing where each node is responsible to store its own checkpointing files. However in this section, we used a soft checkpointing where the initiator it selected using our proposition in the previous section. We also implement an I/O manager in the initiator to improve its role in the checkpointing process and reduce the checkpointing overhead caused by the storage time. The soft checkpointing can be used with any coordinated checkpointing strategy (all/Min and blocking/non blocking checkpointing).

In the soft checkpointing, the initiator collects the checkpointing files of all the nodes involved in the checkpointing process and ensures the checkpointing atomicity. The atomicity means that all the nodes concerned by the checkpointing have successfully create theirs checkpointing file. If any node (concerned by the checkpointing) has failed to create or send its file to the initiator, the initiator will cancel the actual checkpointing to preserve the consistency.

After the initiator ensures checkpointing atomicity of its nodes, its storage manager handles the I/ O management to minimize the checkpointing latency. The storage manager is similar to ROMIO. But in ROMIO, the Data Sieving and collective I/O will always be executed regardless of the size of useless data quantity caused by Data Sieving (Holes). In this case, the amount of unnecessary data can be large compared with the useful data, which increases the cost and the time of the I/O . To resolve this issue, the storage manager of the initiator executes Collective I/O for data requested by different nodes, and then performs Data Sieving only if the size of useless data $D_j$ does not exceed a certain threshold $\alpha_j$ versus the total size of data to be written $D_j$ by it (initiator). The threshold $\alpha_j$ is specified as an input. The $\beta_j$ parameter of the initiator represents the percentage of useless data written relative to the entire data (see Formula 9). The $\beta_j$ will be compared to $\alpha_j$ to decide to perform a Data Sieving or not (simple buffering).

$$\beta_j = (100 \times UD_j)/D_j \quad (9)$$

This strategy eliminates the problem of transfer of large quantity of useless data. The details of storage algorithm executed by the initiator are illustrated in Figure 8.

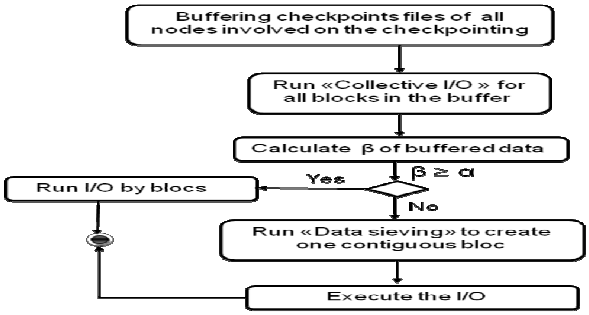


*Fig. 8: Storage Algorithm*

To explain the storage manager of the initiator, we offer the example of Figure 9. In this Figure, there are three nodes. {N1, N2, N3} that have the same initiator. Each node requires a set of blocks of the same file (blocks requested by all these nodes are from the same file).

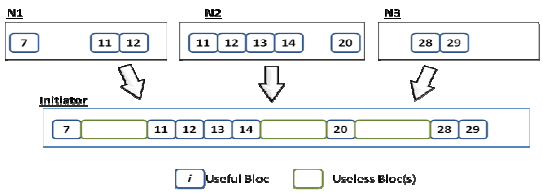


*Fig. 9: Storage scenario Example*

For example, the N1 requests blocks 7, 11 and 12, these blocks are useful. Each node sends its useful blocks to its initiator. The initiator executes the collective I/O in the requested blocks by collecting the useful blocks in the buffer of its memory (union of useful blocks of nodes involved in the checkpointing process). Collective I/O allows to create more contiguous blocks (case of blocks {11, ..., 14} in N2 and N3) and removes redundancies (case of blocks {11, 12} in N2 and N3). In this case, if the block size is 20 bytes so the size of useful data is 20 × 10 = 200 Bytes. In ROMIO, the system executes the Data sieving to create more contiguous blocks. The total data size will be (29-7) × 20 = 440 bytes. But in this case 440-200 = 240 Bytes will be useless data, so ROMIO can increase the checkpointing latency. However, in CSDS: $UD_a$ = 240 Bytes; $D_a$= 440 Bytes and if $\alpha$ =30% (specified by SLA) then $\beta$= $(100\times DI_a)/D_a$ =$(100\times240)/440 \approx 54{,}5\%$). In this case, the storage manager of the initiator decides not to run the Data sieving and sends only useful data blocks {{7}, {11, 12, 13, 14}, {20}, {28, 29}}.

## VI. Experimental Results

It is clear that non-blocking coordinated checkpointing minimizes the overhead of checkpointing but the blocking checkpointing is easier to implement and minimizes the rate of stored data to ensure consistent rollback. For further details, paper [1] presents a comparative study between two strategies for checkpointing.

The aim of the first part of our work is studing the impact of the initiator choice on the four coordinated checkpointing strategies: Min-process blocking checkpointing, Min-process non-blocking checkpointing, All-process blocking checkpointing and All- process non-blocking checkpointing. For the initiator choice, we selected three strategies: LID (Lowest ID), Hpop (Higher popularity) and our approach that we name it simply Our Init. The used parameters in our simulations are presented in Table1.

**Table1**: Simulation parameters

| **Parameter** | **Value** |
|---|---|
| Number of VM per server | 10-100 |
| Server BW | 1 Gega bit per second |
| Cloudlet number (Tasks) | 1500 |
| Cloudlet length | 100-12000 MIPS |
| communication rate $\mu$ | 2-100 |
| Checkpoint interval $CP_{Interval}$ | 100-500 second |
| Failure rate $\lambda$ | 2 to 5 per period |

### *A. Initiator Choice*

The first series of experiments studied the Overhead cau

The first series of experiments studied the Overhead caused by the four checkpointing protocols using several initiator choice strategies. The overhead in this work in presented as the rapport between the response time with and without the checkpointing (See Formula10):

$$Overh = \frac{RT_F}{RT_{\overline{F}}} \quad (10)$$

Where:

- $RT_F$: Response time using a fault tolerance strategy
- $RT_{\overline{F}}$: Response time without using any fault tolerance strategy.

In the case of min-coordinated checkpointing and in both types of this checkpointing: blocking (Figure 10 -a- ) and non-blocking (Figure 10 -b -) we have notice two points:

First, Hpop improves checkpointing (minimizes overhead) with an average of 20%. However, our approach provides an improvement of 29%. The second point is: the impact of choice of initiator is greater in the case of blocking coordinated checkpointing compared to non-blocking coordinated checkpointing with a percentage of 4.6%. In blocking coordinated checkpointing, nodes suspend the execution to record their statements. So the selection of initiator based on its processing speed can minimize the time to create checkpoints.

In the case of ALL-Coordinated checkpointing and in both types of this checkpointing: blocking (Figure 10 -c- ) and non-blocking (Figure 10 -d -) / we notice that Hpop strategy has no impact on checkpointing performance because it is based on the popularity of the node and in case of ALL-checkpointing, the nodes have the same popularity that equal the number of nodes in the system. So Hpop is created for min-coordinated checkpointing. But our strategy has improved in performances by 2.7% because the initiator is the most powerful node in the system in term of speed.

The second series of experiments calculates the rollback cost (See Figure 11). Both types of strategy min blocking / non-blocking use the min rollback [1], and also All-blocking/ non blocking coordinated checkpointing use All-rollback technique. In case of Min-rollback (See Figure 11 –a-), our approach appears effective over other strategies, it avoids the problem of famine by ensuring checkpointing selection according to the node's csn. In case of All-Rollback strategy (See Figure 11 –b-), the initiator choice has no impact on the rollback performances.

During our researches, we found many works proposed and managed the concurrent checkpointing [11][12]. In the concurrent checkpointing, multiple initiators trigger checkpointing at the same time. In the latest round of experiments we studied the impact of checkpointing concurrency in the overload and the rollback cost in the case of min coordinated checkpointing (See Figure 12). Increase the number of initiators by checkpointing increases the overhead and even the cost of recovery in all approaches of initiator selection (See Figure 12 –a-). Unexpected results of concurrent checkpointing in case of rollback (See Figure 12 –b-) due to the relation between the initiators in each checkpointing round. It is possible to improve the performance

of our approach and other approaches even in case of recovery if insured by concurrent initiators are totally independent of each other directly and transitively. The independence condition requires centralized decision otherwise the overload checkpointing become unbearable.

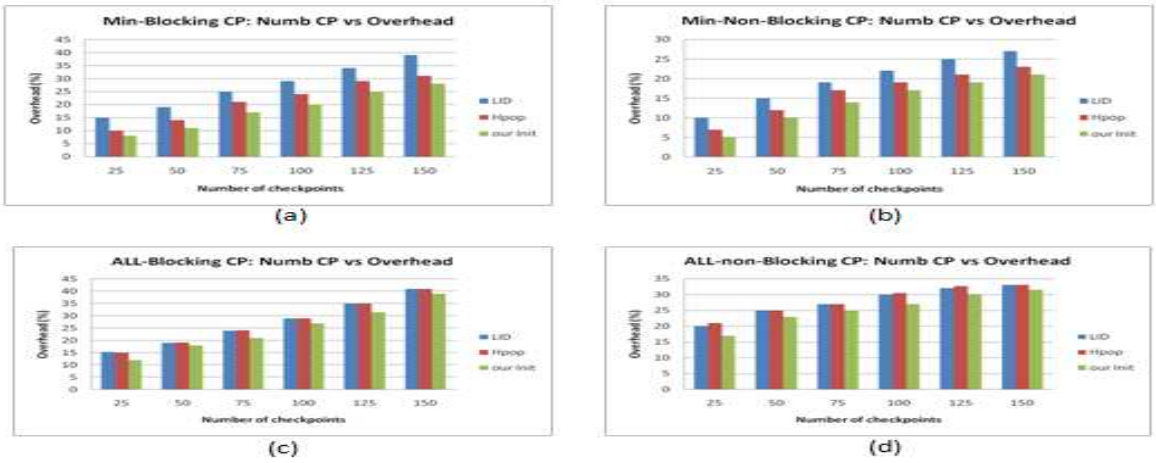


*Fig. 10: Overhead vs Number of checkpointing*

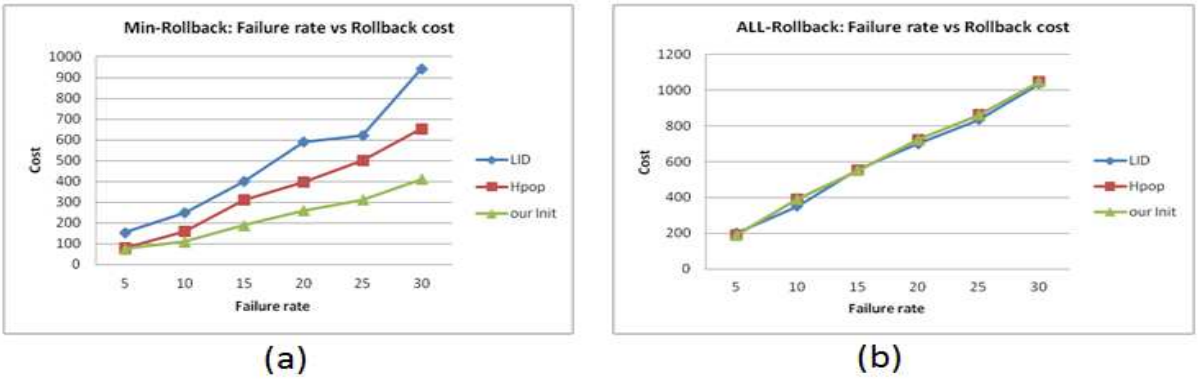


*Fig. 11: Rollback cost Vs failure rate.*

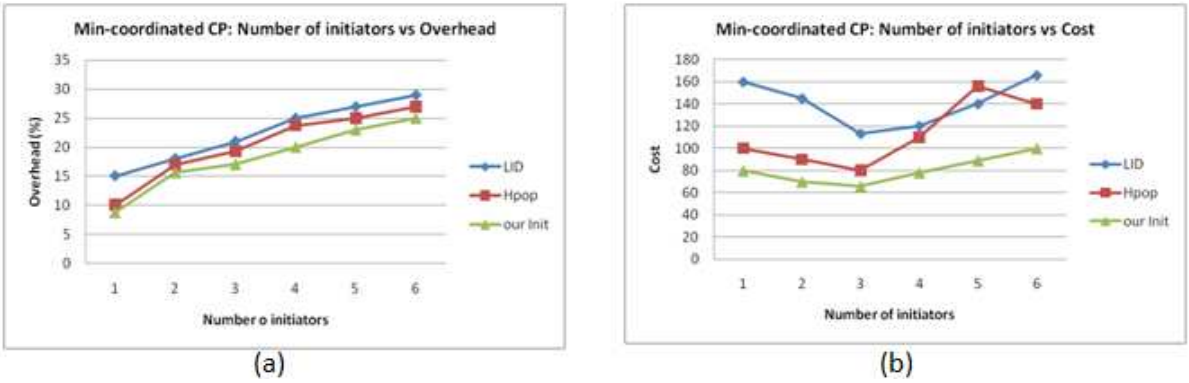


*Fig. 12: Impact of Number of initiators on cost and overhead*

### B. *Soft Checkpointing*

All previous experiments were carried out with a hard checkpointing (without the use of soft checkpointing), that is to say that the initiator manages just the checkpointing and each node is responsible for storing its own checkpointing file. The goal was: to measure the minimum impact on the initiator checkpointing without assigning the other roles.

In this part of the experiments, we focused on the soft checkpointing in minimum coordinated checkpointing and we used the same parameters of Table1. According to the results of comparing hard and soft checkpointing, we noticed that the initiator impact on checkpointing increases by almost 17% compared to a hard checkpointing whatever the strategy of initiator choice. It is clear that in case of all coordinated checkpointing; the impact of soft checkpointing will be bigger (See Figure 13).

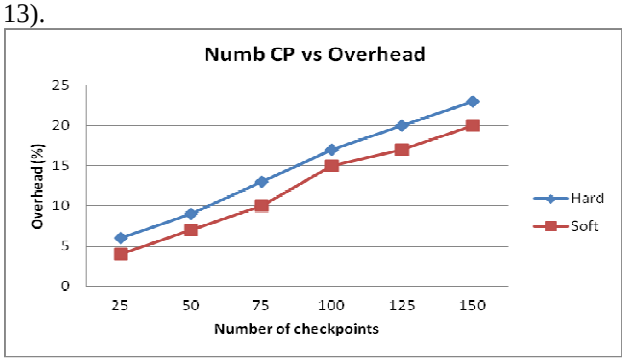


*Fig.13: Overhead in soft and hard checkpointing*

We have strengthened the role of initiator in the soft checkpointing by the Storage Service (CS + SS) and compared its performance with the soft checkpointing without checkpointing Storage Service (SC). Both (SC + SS) and (SC) select their initiators using our contribution1.

The first experiment in this part is destined to measure the overhead in case of different sizes of checkpointing files. According the results illustrated in Figure 14-a-, the Overhead time increases if the size of files increases because of the storage time. However, our strategy CS + SS is better than CS because the storage service reduces the transfer of useless data during the I/O.

The goal of the second experiment is measuring the impact of the number of nodes involved in the checkpointing on the overhead caused by (SC + SS) and (SC) .the results (See Figure 14-b-) prove that increasing the number of the nodes concerned by the checkpointing process increases automatically the overhead because the number of checkpointing files will increase. However in CS + SS, the storage service in the initiator uses the collective I/O to collect and organise data and it uses also smart data sieving to reduce the transfer time.

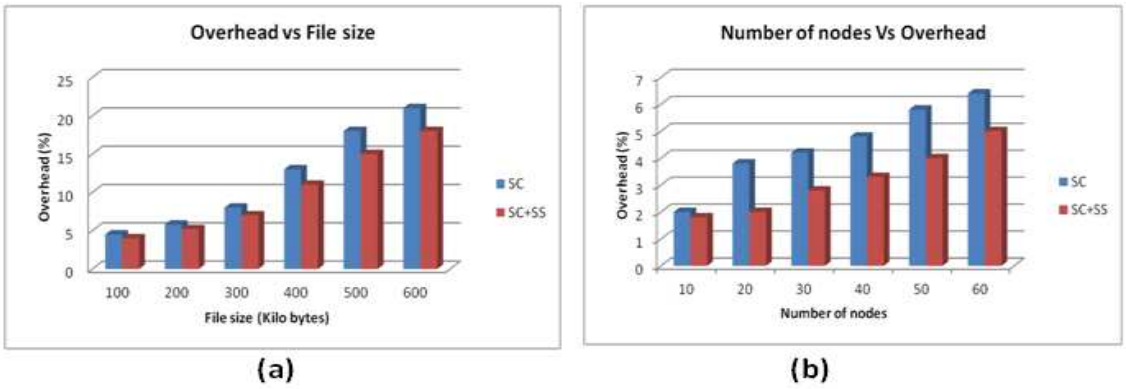


*Fig.14: Overhead in soft checkpointing with and without storage service*

## VII. Conclusion

The checkpointing is a high performance tool to ensure fault tolerance and system reliability. The literature offers many checkpointing protocols that ensure the creation of a coherent state for the rollback. In this paper, we have explained in details the most popular coordinated checkpointing strategies and the I/O techniques. Then we proposed two contributions. In the first contribution, we studied the impact of initiator choice on these protocols. We also proposed a strategy for the selection of initiator that accelerates checkpointing and minimizes the rollback cost. The second contribution proposes a soft checkpointing with a storage service based on collective I/O and smart data sieving. The experimental results prove that:

- The strategy of initiator choice has a non-negligible impact on checkpointing performances, especially in case of blocking coordinated checkpointing.
- Considering the physical characteristics of initiator (speed, overhead, ...) reduces the checkpointing overhead.
- Decreasing the gap between checkpoint sequence numbers improve greatly the performances of minimum rollback strategies.

- The initiator choice has no impact on all-rollback strategies.
- The concurrent checkpointing increases the checkpointing overhead. And does not necessarily improve the rollback performances.
- The concurrent checkpointing can improve recovery performance if the initiators are completely independent with minimum csn.
- The initiator choice has higher impact in case of soft checkpointing compared to hard checkpointing.
- The soft checkpointing can improve the checkpointing performances.
- Using I/O management in soft checkpointing reduces the storage time of checkpointing and therefore it reduces also the overhead.
- Using a smart data sieving reduces the transfert of useless data during the storage.

In the future works, we will use a smart strategy for the initiator selection by using the techniques of consensus between nodes represented by agents. We will also improve the I/O technique using other parameters to balance between the collective I/O with or without data sieving such as cost and consumed energy during the I/O.

## REFERENCES


[1] D. Buntinas, C. Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, Franck Cappello: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols, Future Generation Computer Systems, Vol.24, No.1, 2008, pp. 73-84

[2] K.M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, Transactions on Computer Systems, Vol.3, No.1, 1985, pp. 63–75.

[3] H. Hui, Z. Zhan, W. Bai Ling, Z. De Cheng and Y. Xiao-Zong, A Two-level Application Transparent Checkpointing Scheme in Cloud Computing Environment, International Journal of Database Theory and Application, Vol.6, No.2, 2013, pp. 61-71.

[4] M Slawinska, J Slawinski and V.Sunderam, Unibus: Aspects of heterogeneity and fault tolerance in cloud computing , In Proceeding of IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW), 2010, pp. 1-10.

[5] T. Yuval and C.H. Séquin. Error recovery in multi-computers using global checkpoints. In Proceeding of International Conference on Parallel Processing, 1984, pp. 32–41.

[6] Koo R. and Toueg S., Checkpointing and Roll-Back Recovery for Distributed Systems, IEEE Trans. on Software Engineering, Vol.13, No.1, 1987, pp. 23-31.

[7] Cao G. and Singhal M., On coordinated checkpointing in Distributed Systems, IEEE Transactions on Parallel and Distributed Systems, Vol.9, No.12, 1998, pp. 1213-1225.

[8] N. Limrungsi, J. Zhao, Y. Xiang, T. Lan, H. H. Huang and S. Subramaniam, Providing reliability as an elastic service in cloud computing, In Proceeding of IEEE International Conference on Communications (ICC12), 10-15 June 2012, Ottawa, Canada, pp. 2912-2917.

[9] R. Prakash and M. Singhal. Low-Cost Checkpointing and Failure Recovery in Mobile Computing Systems. IEEE Transactions on Parallel and Distributed Systems, Vol.7, No.10, 1996, pp.1035–1048.

[10] P. Kumar, P. Gahlan, A Low-Overhead Minimum Process Coordinated Checkpointing Algorithm for Mobile Distributed System, International Journal of Computer Applications, Vol.3, No.1, 2010, pp. 17-21.

[11] G. Cao, M. Singhal, Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing systems, IEEE Transaction On Parallel and Distributed Systems, Vol.12, No.2, 2001, pp. 157-172.

[12] Chaoguang Men, Xiaozong Yang , Using Computing Checkpoints Implement Consistent Low-Cost Non-blocking Coordinated Checkpointing, Parallel and Distributed Computing: Applications and Technologies Lecture Notes in Computer Science Vol.3320, 2005, pp. 570-576 .

[13] J. Surender, S. Arvind, K. Anil and S. Yashwant, Low Overhead Time Coordinated Checkpointing Algorithm for Mobile Distributed Systems, Computer Networks & Communications (NetCom) in Lecture Notes in Electrical Engineering, 131(2013) pp. 173 -182.

[14] Y. Liu, W. Wei and Y. Zhang, Checkpoint and Replication Oriented Fault Tolerant Mechanism for Map Reduce Framework, Telkomnika Indonesian Journal of Electrical Engineering, Vol.12, No.2, 2014, pp.1029-1036.

[15] Z. Abdelhafidi, M. Djoudi, M.B. Yagoubi An Improved schema of coordinated checkpointing protocol for distributed systems based on popular process. International Conference on Innovations in Information Technology (IIT), 2012, pp. 367-372.

[16] Ouyang, X., K. Gopalakrishnan and D-K. Panda. 2009. Accelerating checkpoint operation by node-level write aggregation on multicore systems. In proceeding of International Conference on Parallel Processing (ICPP'2009), Vienna, Austria, 22-25 September 2009, pp : 34-41.

[17] Cornwell, J. and A. Kongmunvattana, 2011. Optimized I/O Operations for Checkpoint Creation in BLCR", 24th International Conference on Computers and Their Applications in Industry and Engineering (CAINE'11) Honolulu, Hawaii, USA 16-18 November 2011. pp. 284-289.

[18] Meroufel, Bakhta; Belalem, Ghalem, Lightweight coordinated checkpointing in cloud computing, Journal of High Speed Networks , Volume 20 (3): 131-143, 2014

[19] Jose Carlos Sancho, Fabrizio Petrini, Greg Johnson, Juan Fernandez, and Eitan Frachtenberg. On the feasibility of incremental checkpointing for scienti_c computing. In 18th International Parallel and Distributed Processing Symposium (IPDPS), page 58, 2004.

[20] Chen, Y., X.-H. Sun, R. Thakur, P. C. Roth and W. D. Gropp, 2011. LACIO: A New Collective I/O Strategy for Parallel I/O Systems, IEEE International Parallel & Distributed Processing Symposium (IPDPS), Anchorage, Alaska 16-20 May 2011, pp: 794-804.

[21] Del Rosario, J., R. Bordawekar and A. Choudhary. 1993. Improved parallel I/O via a two-phase runtime access strategy.ACM SIGARCH Computer Architecture News - Special issue on input/output in parallel computer systems. 21(5): 31-38.

[22] Thakur, R., W. Gropp and E. Lusk, 1999. Data Sieving and Collective I/O in ROMIO. In Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation, February 1999, pp:182–189.

[23] Thakur, R., W. Gropp, and E. Lusk. 1999b. On Implementing MPI-IO Portably and with High Performance. In Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems, pp: 23-32

[24] Cornwell, J. and A. Kongmunvattana, 2011. Optimized I/O Operations for Checkpoint Creation in BLCR", 24th International Conference on Computers and Their Applications in Industry and Engineering (CAINE'11) Honolulu, Hawaii, USA 16-18 November 2011. pp. 284-289.