Service to Fault Tolerance in Cloud Computing Environment

BAKHTA MEROUFEL
Department of Computer Science
Faculty of Exact and Applied Sciences
University of Oran 1, Ahmed Ben Bella
BP. 1524, EL M'Naoeur, Oran, 31000
ALGERIA
bakhtasba@gmail.com

GHALEM BELALEM
Department of Computer Science
Faculty of Exact and Applied Sciences
University of Oran 1, Ahmed Ben Bella
BP. 1524, EL M'Naoeur, Oran, 31000
ALGERIA
ghalem1dz@gmail.com

Abstract: Cloud Computing refers to the use of memory and computing capabilities of computers and servers around the world, so the user has considerable computing power without the need for powerful machines. The probability of failure occurring during the execution becomes stronger when the number of nodes increases; since it is impossible to fully prevent failures, one solution is to implement fault tolerance mechanisms. In this work, we have developed a fault tolerant service based on the checkpointing in cloud computing. Our fault tolerance service uses a semi-coordinated checkpointing that minimizes the consumed energy and the overhead by decreasing the time of the coordination phase. Our service also decreases the rollback cost. The experimental results show the effectiveness of our proposition in term of execution time, energy consumption of SLA violation.

Key–Words: Fault tolerance, Cloud computing, Virtualization, Checkpointing, Overhead, Rollback, Coordination, Recovery line, SLA violation, I/O.

1 Introduction

Cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized computers. The latter is dynamically provisioned and represented as one or more unified computing resources based on service level agreement (SLA) established through the negotiation between the service providers and consumers. Cloud computing is a term that involves delivering hosted services over the Internet. By using the virtualization concept, cloud computing can also support heterogeneous resources and achieved flexibility. Another important advantage of cloud computing is its scalability. Cloud computing has been under growing spotlight as a possible solution for providing a flexible on demand computing infrastructure for a number of applications. All these factors increase the popularity of cloud computing. The services are broadly divided into three categories Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [1]. On the other hand, the reliability of Cloud computing still remains a major concern among users. Due to economic pressures, these computing infrastructures often use commodity components exposing the hardware to scale and conditions for which it is not originally designed [2]. As a result, significantly large number of failures manifest in the system and seemingly imposes high implications on the hosted applications, impacting their availability and performance. For example, a computing system with 200,000 nodes will experience a mean time between failures (MTBF) of less than one hour, even when the MTBF of an individual node is 5 years [3]. In this context, the applications require fault tolerance abilities so that they can overcome the impact of system failures and perform their functions correctly when failures take place [4]. In this case, the fault tolerance is an important issue to ensure the cloud reliability; however, the choice of fault tolerance technique remains a critical problem especially with the SLA and energy parameters. In this paper, we provide the fault tolerance as a SaaS based on the checkpointing technique. In the checkpointing, the system records its state in checkpoints files. When a task fails, it is restarted, instead of initiating from beginning, from the recently checked pointed state. The checkpointing is widely used in the cloud environment [5, 6, 7, 8, 9]. And beside the fault tolerance, the checkpointing can be used for other important purposes such as: migration, debugging, load balancing or even improving other fault tolerance strategies [10, 11]. The goal of the checkpointing is creating a consistent state stored in a stable memory [9, 12]. A consistent state can not contain any orphan messages. And the state is strongly consistent if it is transit message free (beside the orphan messages) [13]. The orphan message is a message whose its *receive event* is recorded in the checkpointing file,

but its *send event* is not recorded. In case of the recovery, the destination node would receive this message twice, which could result in unpredictable application behaviour. The transit message is a message whose its *send event* is recorded, but its *receive event* is not i.e. the message is lost after the rollback. To create a consistent state; the literature proposes two different checkpointing categories [14]:

- Coordinated (synchronized) checkpointing: the nodes (where the tasks are executed) communicate with each other and coordinate their efforts to create a consistent state.
- Uncoordinated (independent) checkpointing: when the checkpointing interval expires, each node records its state independently from the other nodes.

Considering the cloud characteristic and each checkpointing category, the independent checkpointing is scalable, but it is cannot ensure a consistent state and it increases considerably the overhead because of the domino effect. In a domino effect, unbounded, cascading rollback propagation can occur during the process of finding a consistent global checkpoint, which makes the recovery cost unacceptable and garbage collection complex be implemented [15]. The coordinated checkpointing ensures the desired consistency; however, it is very expensive in term of overhead and energy consumption because of the coordination among the system nodes [16]. Our paper improves the classical coordinated checkpointing to ensure a strong consistency with the minimum overhead by minimizing the number of participants in the coordination phase and the rollback process. The classical coordinated checkpointing forces all the VMs (Virtual Machines) to coordinate and record their states. In case of failure detection, all those VMs will execute a rollback to ensure the strong consistency. The classical coordinated checkpointing characteristics make it inappropriate for the cloud computing since it consumes more resources and energy. It also increases considerably the overhead which makes the user unsatisfied with the system performances. However, in our approach, only the VMs communicating with the initiator during the last checkpointing interval will be involved in checkpointing creation and in case of failure, only the VMs communicating with the failed VM during the last checkpointing interval will roll back to their last states. In this case (our approach), the problems of classical checkpointing will be eliminated or at least reduced. Our strategy is also fault tolerant which decreases the damages in case of checkpointing failure or cancellation.

The remainder of the paper is organized as follows. Section 2 reviews some related works. Section 3 describes the system model and explains in details our fault tolerance service based on semi-coordinated checkpointing. Section 4 shows some experimental results and analysis of performance of the proposed approach. Finally, a summary and some perspectives are given in Section 5.

2 Related works

In a cloud environment, the checkpointing is used for many purposes such as: fault tolerance, migration, load balancing and debugging. Since the major problem of the checkpointing is the overhead, several works propose many techniques dealing with this problem. We can distinguish three axes of researches: checkpoints sizes, number of checkpointing and checkpointing protocol. The first axe aims at reducing the checkpointing size. By decreasing the amount of data recorded during the checkpointing, the storage time and the I/O overhead will be also decreased. The paper [17] uses the incremental checkpointing where only those memory pages (or VM data chunks) that are modified over the time interval will be saved. The compression is also another technique reducing the size of checkpointing files [19].

The second axe's goal is minimizing the number of checkpoints by optimizing the checkpointing interval. The paper [19] proposes a checkpointing interval based on the overhead and the mean time to interruption (MTTI). The interval used in [19] is a reference for many works in this field [20]. It uses the failure rate to control the checkpointing frequency. However, in the cloud computing other critical parameter which is the SLA violation should be considered. In [6], an authors use two kinds of thresholds: price threshold and time threshold to optimize the checkpointing interval. In this strategy, a coordinator manages the communication between the user and the instance to decrease SLA violation. The work in [6] uses the price history of spot instances and the desired task reliability to make decisions about when the checkpointing should be executed.

The last researching axe focuses on the improvement and the optimization of checkpointing protocol. Beside the overhead, the consistent state should be ensured by using checkpointing protocols such as: coordinated, uncoordinated or hybrid strategies. Since each strategy has its negative effects, it must be improved or combined with other techniques. The checkpointing technique is widely studied and used in a distributed systems (grid computing and mobile networks). The paper [21] proposes a coordinated

checkpointing in distributed environment. It suggests blocking the communication during the checkpointing creation to ensure a consistent state (a state without any orphan messages). However, in this work all the processes participate in the checkpointing which increases the overhead and the blocking time. In order to limit the number of nodes involved in the coordination, the authors of [15] use a Quasi-Asynchronous protocol in mobile networks. The proposed protocol coordinates between only the communicating nodes during the last checkpointing interval; the other nodes create their checkpoints in an asynchronous way. But in the work [22], the authors describe some scenarios to prove that the Quasi-Asynchronous protocol can create inconsistent states. Other protocols use the time based coordinated checkpointing to decrease the coordination rate during the state recording, such as the papers [23, 24, 25, 26, 27, 28]. In time based checkpointing, it is assumed that the clocks are approximately synchronized with a certain drift. So when the checkpointing interval expires, all the nodes create their checkpoints at the same time and the consistency is ensured by blocking the communication during the clock/timer drifts. However, the re-synchronization cost is very expensive [28]. The non-blocking coordinated checkpointing is proposed in [29]: it does not block the communication during the checkpointing, but deals with the orphan messages by storing and piggybacking them with a lot of data. Despite that the checkpointing is widely used in cloud computing, the majority of existing works focus only on the first and second axes of researches (size and interval of checkpointing). In [30, 31], the authors used the uncoordinated checkpointing to ensure the fault tolerance without considering the VMs communications created by the task dependencies. In this case, the possibility of creating an inconsistent state exists and the cost of the rollback can be very expensive because of the domino effect. The paper [7] creates a new topology called VIOLIN that uses a classical coordinated checkpointing [29]. However, the used strategy suffers from many problems such as: the need for a high rate of coordination to ensure consistency and termination detection. The only objective of paper [7] is proving that the classical checkpointing strategies used in other distributed systems can be used in the cloud environment. Unibus proposed in [2] uses Distributed Multi-Threaded CheckPointing (DMTCP). In DMTCP, all the processes are blocked until the storage of checkpoint files, then they resumed their work. In addition, DMTCP in [2] is applied in all cloud level (process, VM and servers) which decreases scalability.

3 Our Contribution

The cloud applications (such as e-commerce and scientific applications) consist typically of several tasks, which communicate with each other [31]. A task consists of some computation and communication phases. Since the tasks are executed in VMs, the communication between tasks is executed by their VMs. In this case, the transit and orphan messages can exist during the creation of checkpoints. Most existing works on checkpointing in cloud computing focus on the checkpointing size or the checkpointing interval rather than the checkpointing technique itself. In this paper, we propose a fault tolerance service based on Semi-Coordinated Checkpointing (SCC). The system model is a set of servers (Hosts) that contain many virtual machines (VM). The user's job will be divided into tasks. These tasks are deployed on m virtual machines running at n servers, each VM is running one task at the moment. The servers or the VMs are in accordance with fail-stop model [16].

3.1 System Model

To ensure the fault tolerance, we added a fault tolerance service based on the checkpointing (See Figure 1). This service is implemented inside each data center and it contains four modules (sub-services):

- Checkpoint module: this module is responsible for recording the system state. When the checkpointing interval T_{CP} expires, the checkpointing module selects an initiator VM_{init}. When the involved VMs in the checkpointing are selected, this module records their states by saving all the data necessary to ensure the consistent state and the correct rollback in case of failure. The checkpoint files are stored in a stable memory.

- Supervisor module: this module ensures the atomicity and the correctness of the checkpointing.

- Error processing module: This module captures the system state by detecting the failed VM or host. When failure appears, this module selects the checkpoint file in question and triggers the rollback module. The error processing module can also control the checkpointing interval T_{CP} depending on the failure rates (adaptive checkpointing interval).

- Rollback module: this module ensures that the failed VM will resume its execution correctly using the checkpointing file selected by the Error processing module.

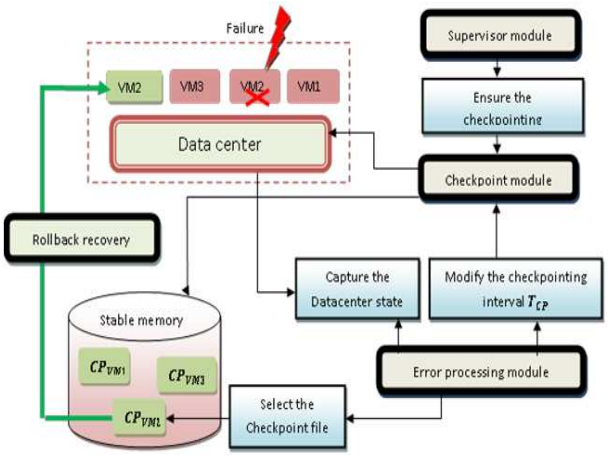

Figure 1: Fault Tolerance Service Architecture

In Figure 1, VM2 fails, so it will be replaced by another VM. This new VM will resume VM2 execution using the checkpoint file CP_{VM2}.

3.2 Checkpointing Protocol

The independent (uncoordinated) checkpointing does not need any coordination among VMs; however, it causes an extra-overhead during the rollback recovery because of the domino effect. It also needs to store all the checkpointing files during the job life time, which increases the resource consumption. The coordinated checkpointing ensures consistency and it needs to store only the last checkpoint file in the stable memory. However, the coordination phase increases the overhead. The creation of a strong consistent state means the elimination of any orphan or transit messages. These types of messages (transit/orphan) can be created only between the communicating VMs. The VMs that do not communicate with each other cannot create transit or orphan messages because there are no message exchanges. On the basis of this observation, our Semi-Coordinated Checkpointing (SCC) reduces the number of VMs involved in the checkpointing by forcing only the communicating VMs during the last checkpointing round to create their checkpoints or to rollback, the other VMs continue their execution. The SCC is also two phases blocking coordinated checkpointing. SCC phases are the creation of tentative checkpoints (TCP) and then convert them to permanent checkpoints (PCP). The TCP is a checkpoint file stored in the local memory. The PCP is a TCP stored in the stable memory. The goal of the two phases is reducing the damages in case of checkpointing cancellation or failure. This means that if the checkpointing is failled or cancelled, the VMs cancel only the local TCP stored in the local memory without affecting the consistency of the last checkpoint file stored in the stable memory and without any I/O overhead (if the system uses only PCP, it will execute I/O to eliminate this PCP from the stable memory and the last checkpoint will be unavailable). SCC ensures also a strong consistency by:

- Ensuring that the orphan messages cannot be created during the checkpointing. This can be done by blocking the communication during the checkpointing process. Blocking the communication in our approach means just freezing this communication, i.e. during the checkpointing, the VM continues its execution until it finds a Send or a Receive event.

- Ensuring that the transit messages are all stored in the checkpoint file. In this case, and during the rollback, the VM can ask the other VMs to re-send to it the transit messages.

To decrease the overhead caused by the coordination phase, we minimize the number of VMs involved in the checkpointing in each period. Each VM uses a Boolean vector to track all the VMs communicating with it during the last checkpointing interval. When an initiator is selected, only the communicating VM with this initiator during the last checkpointing interval will be forced to create its checkpoints. Each time, when the VM creates its checkpoint, it initializes its dependency vector; it will also increment its checkpointing sequence number (csn). The csn is an integer number that will be incremented each time the VM creates its checkpoint file and it is initialized at 0 in the beginning of job execution.

The first step in the SCC algorithm is the selection of the initiator VM_{init}. In many existing works in the literature, the checkpoints approaches do not focus on the choice of the initiator [14, 15, 22]. The paper [32] shows that the initiator choice affects the system performances considerably. In our work, the initiator is the VM with the minimum csn and the best performances (the bandwidth, the processor speed). The first criterion allows the system to avoid the problem of famine (only some VMs create their checkpoints during the job life) and it reduces the csn differences among the VMs. And by choosing the most performing VM (the second criterion) the checkpointing process will be accelerated. After selecting the initiator VM_{init}, the VM_{init} collects the dependency vectors of the VMs and creates the dependency matrix. The dependency matrix identifies all the VMs that have communicated with the initiator directly or transitively during the last checkpointing interval. Those VMs are included in the $ListCP_{init}$ list. The SCC

uses several messages exchanged between the initiator and the concerned $VM_i \in ListCP_{init}$ to ensure the atomicity and the correctness of the checkpointing:

- REQUEST: sent by the initiator VM_{init} to all $VM_i \in ListCP_{init}$.
- RESPONSE: sent by $VM_i \in ListCP_{init}$ to the initiator VM_{init} to confirm the creation of local Tentative Checkpoints TCP. The tentative checkpoints are created in the local memory.
- COMMIT: sent by the initiator VM_{init} to $VM_i \in ListCP_{init}$ when it receives the "RESPONSE" of all the involved VM. The initiator uses a counter N_R to calculate the number of received RESPONSE. When the $V_i \in ListCP_{init}$ receives this message, it converts its tentative checkpoint to permanent checkpoint by storing it (the TCP) in the stable memory.
- ABORT: sent by $VM_i \in ListCP_{init}$ to the initiator to inform it that this VM can not or it refuses to create its checkpoint.
- DISCARD: sent by the initiator VM_{init} to $VM_i \in ListCP_{init}$. When a VM receives this message, it removes its last tentative checkpoint from the local memory. The "DISCARD" message is used to cancel the checkpointing process when the initiator does not receive all the "RESPONSE" of all $VM_i \in ListCP_{init}$ or the initiator has received at least one "ABORT" message.

Figure 2 explains the process of our Semi-coordinated Checkpoint (SCC). SCC algorithm is the following: during the failure free execution, the fault tolerance service (checkpointing module) selects the initiator VM_{init}. When the checkpointing interval T_{CP} expires, the initiator VM_{init} collects the dependency vectors and identifies the participant VMs and add them to the $ListCP_{init}$ list. The VM_{init} sends the "REQUEST" to all the $VM_i \in ListCP_{init}$. When the VM_i receives this message, it can accept or refuse to create its checkpoints. If VM_i accepts the request, it freezes its communication and creates the TCP (tentative checkpoint) then it sends the "RESPONSE" to the initiator VM_{init}. The VM_{init} uses a timeout T_{out} and a counter N_R to calculate the number of received RESPONSE. The VM_{init} compares the N_R with the cardinality of the $ListCP_{init}$, if the initiator has received all the RESPONSE before the T_{out} expires $(N_R = \sharp ListCP(init) \bigwedge T_{out} > 0)$ then the VM_{init} sends to $VM_i \in ListCP_{init}$ a COMMIT message. If VM_i receives the COMMIT message, it stores the

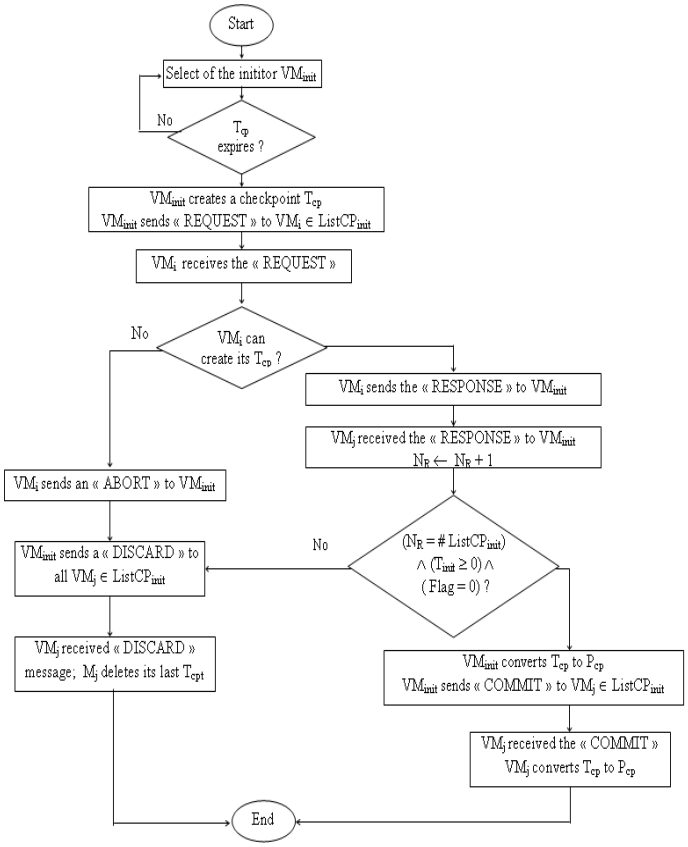

Figure 2: Checkpointing Protocol

TCP in stable memory (PCP). It also continues the task execution and sends the RESPONSE to the initiator. The number of RESPONSE received by the VM_{init} indicates the checkpointing termination.

If a $VM_i \in ListCP_{init}$ cannot create its checkpoints after receiving the REQUEST, it sends the "ABORT" message to the initiator. The initiator uses a Boolean Flag to indicate if a "ABORT" message is received (*Flag=0*) or not (*Flag =1*). In case of receiving the "ABORT" message, and to ensure the checkpointing atomicity, the VM_{init} cancels the checkpointing by sending the DISCARD message to the other $VM_i \in ListCP_{init}$. At the reception of DISCARD message, the VM_i removes the TCP from the local memory without any I/O overhead. The checkpointing can be cancelled if the timeout T_{out} expires before receiving all the RESPONSE (for example, a $VM_i \in ListCP_{init}$ failed during the checkpointing).

The fault tolerance service uses the Error processing module to capture the data center state. In case of failure detection (Host or VM), this module uses the dependency vectors of the other VMs to identify the VMs concerned by the rollback. The concerned VMs are the VMs that have communicated with the failed VM during their last checkpointing interval. The rollback module downloads the checkpoints files from the

Table 1: Simulation Parameters

Parameters	Values
Number of datacenter	1
Number of host	20
Number of Cloudlet	1262
Cloudlet length	120000 bytes
Buffer size	100 Ko (Kilo bytes)
Checkpoint Interval	200 seconds

stable memory and ensures the correct rollback of the concerned VMs.

4 Simulations

In order to evaluate our approach semi-coordinates checkpoint (SCC) and to compare it with a classical coordinated checkpointing approach (CC) proposed in DMTCP [2], we used the well known simulator CloudSim [33]. Several simulation parameters (see Table1) and several scenarios are proposed to experiment the impact of each parameter. In our experimentations, the job is a sequence of tasks and the task is represented by cloudlet.

In this first simulation, we measured the Input / Output during the time in both approaches SCC and CC. The results are shown in Figure 3. We notice that over time, the use of our approach semi-coordinated checkpoint (SCC) reduces *Input/Output* unlike the approach coordinated checkpoint (CC) by almost 49.80%.

The second experimentation studies the scalability by studying the impact of the number of VMs on the I/O overhead. According to the results presented in Figure 4, the number of VMs increases the I/O overhead in both strategies. However, in our approach SCC, the increasing rate is very low. Since SCC is a two phases checkpointing, the TCP minimizes the I/O overhead specially in case of checkpointing failure or cancellation. And since the only communicating VMs create their checkpoints, the number of generated files will be reduced and there for the I/O needed of writing those files will be decreased.

In the third simulation, we measured the overhead caused by each checkpoint approaches during the execution time and with several checkpointing rounds (approximately 120 checkpointing during 10000 seconds). This simulation was performed with the same parameters listed in Table 1. The results are shown in Figure 5. According to the result, the use of our SCC approach reduces the overhead during the time contrary to CC. Therefore, our approach permits to

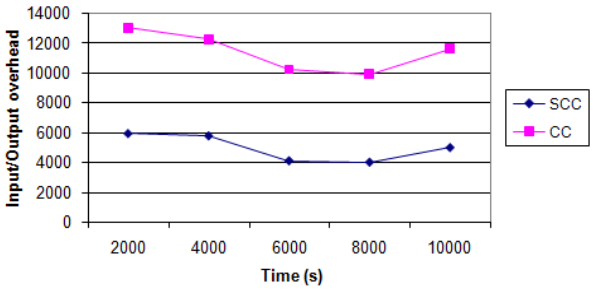

Figure 3: I/O over Time

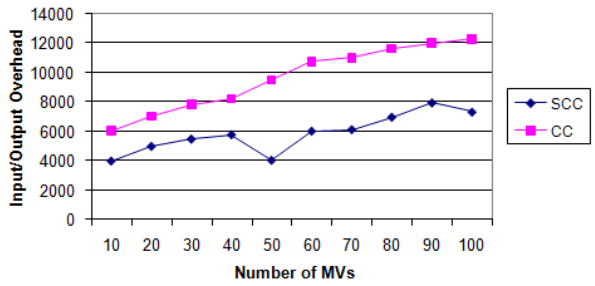

Figure 4: Impact of VMs number on I/O

reduce the load of Hosts and therefore all the system will be less overloaded.

In the fourth experimentation, we vary the number of VMs to test the scalability in term of overhead. Increasing the number of VMs increases always the overhead in both checkpointing strategies (see Figure 6). In this work, the overhead is the difference between an execution time of a system without the fault tolerance service and the execution time of a system that uses our approach of checkpointing during the failure free time (without any failure). In SCC, if the initiator for i^{th} checkpointing round is VM_{init} then the involved VMs will be in the $ListCP_{init}$ list, and $\#ListCP_{init} \leq N$ where N is the number of VMs. So, increasing the number of VMs affects only the time needed to collect the dependency data but the checkpointing itself is related more to the size of $ListCP_{init}$ list.

The communication rate is an important issue to deal with during the checkpointing. So, we proposed the fifth experimentation where we fixed the number of VMs (N) and varied the communication rate. According to the results presented in Figure 7, the communication rate does not infect considerably the CC because all the VMs (N VMs) will be forced to create the checkpoints; and the lazy overhead increasing (See Figure 7) is due to the number of messages needed to be stored in the checkpointing file to ensure a strong consistency (number of transit messages). However, in our approach SCC, it is not the communi-

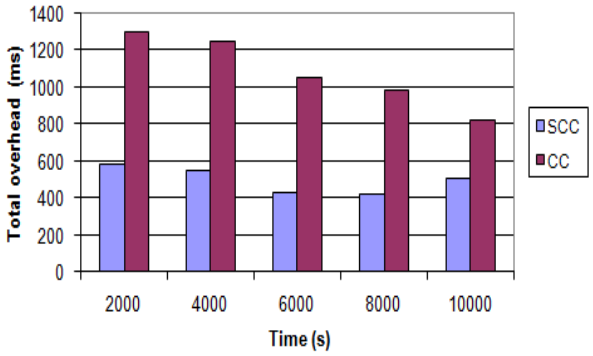

Figure 5: The Overhead over Time

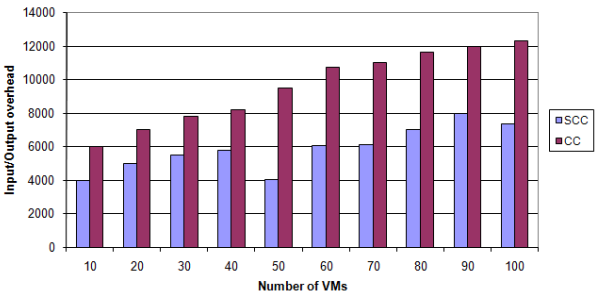

Figure 6: Impact of VMs number on the Overhead

cation rate that infects the performances, but it is the number of different VMs communicating with each other. We can explain the results using two examples:

- In the first example, the communication rate is 200 messages per minute, but only between three different VMs $\{VM_1,VM_2,VM_3\}$. If the VM_1 is chosen as an initiator (VM_{init}=VM_1), so $ListCP_{init}$= $\{VM_2,VM_3\}$ (just two VMs are involved).
- In the second example, the communication rate is only 50 messages per minute but the concerned VMs are $\{VM_1,VM_2,VM_3,\ldots,VM_{25}\}$. If VM_{init}=VM_1, then 24 other VMs will be involved in the checkpointing ($ListCP_{init}$= $\{VM_2,VM_3,\ldots,VM_{25}\}$).

In the sixth simulation, we measured the energy consumed during the checkpointing for the two approaches. The results of this simulation are shown in Figure 8. We notice that the use of our approach SCC reduces the energy consumption during the time contrary to CC approach. The reason of this result is explained in previous experimentations (less overhead and less I/O). So, we can classify it among the approaches of IT Green.

In the cloud computing, SLA (Service Level Agreement) is a very important and critical metric; it

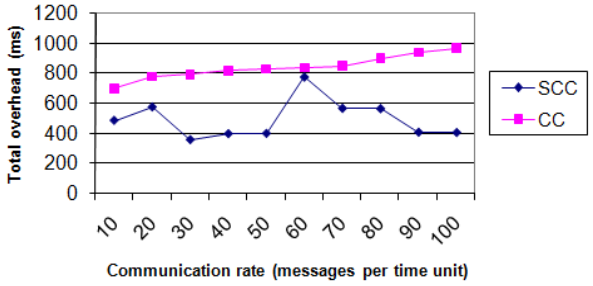

Figure 7: Impact of the Communication rate on the Overhead

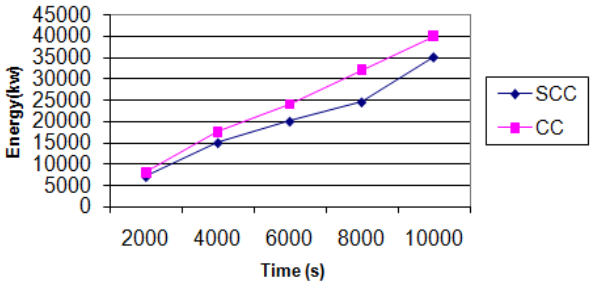

Figure 8: Energy Consumption over Time

represents the satisfaction degree for the user to the services offered by the cloud. In our system model, we suppose that the user can accept some violation threshold and our goal is not exceeding this threshold. In the seventh simulation, we measured the SLA violation caused by both checkpointing strategies. $SLA_{violation}$ caused by the checkpointing service and λ failure rate is calculated using Formula 1:

$$SLA_{violation}(CP) = 1 - \frac{SLA_{CP(\lambda \neq 0)}}{SLA_N} \quad (1)$$

And SLA violation ($SLA_{violation}$) caused by a system without a checkpointing service and λ failure rates.

$$SLA_{violation}(\bar{CP}) = 1 - \frac{SLA_{\bar{CP}(\lambda \neq 0)}}{SLA_N} \quad (2)$$

Where:

- $SLA_{\bar{CP}(\lambda \neq 0)}$ is SLA resulted from a system with λ failures and without using any fault tolerance service.
- SLA_N : is SLA resulted from a normal execution without a fault tolerance service and without any failure (λ=0). SLA_N starts from 100% and it decreases each violation until the end of the job

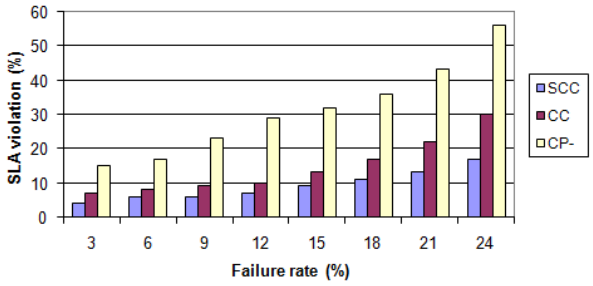

Figure 9: Impact of Failure Rate on the SLA Violation

execution. So, if $?SLA?_N$=60% then the user satisfaction can be estimated at 60%.

- $SLA_{CP(\lambda)}$: is SLA resulted from a system that use a checkpointing strategy with the failure rate λ.

For example, if SLA_N=90% and SLA_CP=70% then: $SLA_{Violation}$=1-70/90 $\cong$ 23%
Formula 1 gives an idea about the SLA overhead using the checkpointing compared to SLA resulted from a normal execution without failure. The result presented in Figure 9 summarizes SLA_{CP} caused by: SCC and CC. We also calculated the $SLA_{violation}(\bar{CP})$ (the violation without using any checkpointing technique). The $SLA_{violation}\bar{CP}$ is very high since the system does not use any fault tolerance service, so the tasks cannot be executed correctly in case of failure. SLA violation increases in both SCC and CC. The SCC reduces the number of involved VMs in the checkpointing and it selects the initiator based on some criteria, so its $SLA_{violation}(CP)$ is less than $SLA_{violation}(CP)$ caused by CC.

5 Conclusion, Limitations and Future Research

In this paper, we have proposed a fault tolerance service in cloud computing based on the checkpointing. Our checkpointing strategy named Semi-Coordinated Checkpointing (SCC) is two phases blocking coordinated checkpointing. It improves the classical coordinated checkpointing by reducing the number of VM involved in the checkpointing and the rollback. The SCC is fault tolerant and scalable, which makes it adequate for the cloud environment. The experimental results prove that the SCC decreases considerably the overhead and SLA violation, compared to the classical coordination checkpointing which makes it contagious for the users. On the other hand, SCC decreases also the energy and resource consumption, which makes it the best choice for the cloud provider.

Nevertheless, the proposed approach is far from complete, we can mention some limitations:

- Currently our proposal has been tested on the simulator CloudSim, we have no guarantee of its behavior in a real environment such as Eucalyptus [34], for example;
- With the current version presented in this paper, we found that our proposal does not seek to balance the physical machine charges;
- A complementary of our approach, we wish to extend a replication service to improve its performance advantage, quality of service and energy consumption;
- During the different simulations launched to study the behavior of our approach, we found that the number of backups and restorations information about check pointing in the stable memory requires a very thorough study, it is the reason that led us to take this into account in our work in parallel.;
- We also found during our experiments that the choice of the initiator of the approach of check pointing affect performance either response time or energy consumption.

References:

[1] M-N.O. Sadiku, S.M. Musa, O. D. Momoh : Cloud computing: Opportunities and challenges. Potentials, IEEE, Vol. 33, 2014, No. 1, pp. 34–36.

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

[3] B. Mills, T. Znati, R. Melhem : Shadow Computing: An Energy-Aware Fault Tolerant Computing Model. International Conference on Computing, Networking and Communications (ICNC'2014), Honolulu, Hawaii, USA, February 3–6, 2014. Pages: 73–77.

[4] R. Jhawar, V. Piuri, M. Santambrogio : A comprehensive conceptual system-level approach to fault tolerance in Cloud Computing. IEEE International Systems Conference (SysCon'2012), 2012; Pages: 1–5.

[5] O. K. Sahingoz, A.C. Sonmez : Agent-based fault tolerant distributed event system. Computing and Informatics, Vol. 26, 2007, No. 5, pp. 489?-506.

[6] D. Jung, S. Chin, K. Chung, H. Yu, J. Gil, An Efficient Checkpointing Scheme Using Price History of Spot Instances in Cloud Computing Environment, Network and Parallel Computing, Lecture Notes in Computer Science; Volume 6985, 2011, pp. 185–200.

[7] A. Kangarlou, P. Eugster, D. Xu : VNsnap: Taking Snapshots of Virtual Networked Infrastructures in the Cloud. IEEE Transactions on Services Computing, Vol. 5, 2012, No. 4, pp. 484–496.

[8] J. Liao : A New concurrent checkpoint mechanism for embeded multi-core systems. Computing and Informatics, Vol. 31, 2012, No. 3, pp. 693-709.

[9] R. Jhawar, V. Piuri : Fault tolerance and resilience in cloud computing environments. In Computer and information security Handbook ;2nd Edition, J.Vecca (Ed.), Morgan Kaufmann, 2013.

[10] 15. D. Sun, G. Chang, C. Miao, X. Wang, Analyzing, modeling and evaluating dynamic adaptive fault tolerance strategies in cloud computing environments, The Journal of Supercomputing, Vol. 66, 2013, No. 1, pp. 193–228.

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

[12] B. Medeiros, J. L. Sobral : Aspect grid : Aspect-oriented fault tolerance in grid plateforms. Computing and Informatics, Vol. 31, 2012, No. 1, pp. 89-101.

[13] R. Koo, S. Toueg : Checkpointing and rollback-recovery for distributed systems. IEEE Trans. Software Eng, Vol. 13, 1987, No. 1, pp. 23–31.

[14] L. Lamport : Time, clocks and the ordering of events in a distributed system. Communications of the ACM, Vol. 21, 1978, No. 7, pp. 558–565.

[15] P.-K. Suri, M. Satiza : An efficient checkpointing protocol for mobile distributed systems. International Journal of Latest Research in Science and Technology, Vol. 1, 2012, No. 2, pp. 109–114.

[16] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, A. Selikhov : MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. Editor : C.G. Roscoe Giles and A. D. Reed and K. Kelley. Proceedings of the ACM/IEEE conference on Supercomputing (SC'2002), Baltimore, Maryland, USA, November 16–22, 2002. Pages : 1–18.

[17] B. Nicolae, F. Cappello : BlobCR: Efficient checkpointrestart for HPC applications on IaaS clouds using virtual disk image snapshots. Editor: S. Lathrop and J. Costa and W. Kramer; Conference on High Performance Computing Networking, Storage and Analysis (SC'2011), Seattle, WA, USA, November 12–18, 2011.

[18] J. S. Plank, J. Xu, R. H. B. Netzer : Compressed differences: an algorithm for fast incremental checkpointing, Univ. of Tennessee, Tech. Rep. CS–95–302, August 1995.

[19] J-W. Young : A first order approximation to the optimum checkpoint interval. Communications of the ACM, Vol. 17, 1974, No. 9, pp. 530–531.

[20] S. Di, Y. Robert, F. Vivien, D. Kondo, C.-Li Wang, F. Cappello : Optimization of cloud task processing with checkpoint-restart mechanism. Editor: W. Gropp, S. Matsuoka. International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13), Denver, CO, USA, November 17–21, 2013.

[21] T. Yuval, C.H. Squin : Error recovery in multicomputers using global checkpoints. In Proceeding of International Conference on Parallel Processing, 1984; Pages 32-41.

[22] G. Cao, M. Singhal :On the Impossibility of Min-Process Non-Blocking Checkpointing and An Efficient Checkpointing Algorithm for Mobile Computing Systems. In Proceeding of the International Conference on Parallel Processing (ICPP'98), Minneapolis, Minnesota, USA, 10–14 August 1998. Pages : 37–44.

[23] C. Flaviu, J. Farnam : A times tamp-based checkpointing protocol for long-lived distributed computations. In Proceeding of Tenth Symposium on Reliable Distributed Systems, September 30 - October 2, 1991, Pisa, Italy (SRDS'91). Pages: 12–20.

[24] C.-Y. Lin, S.-C. Wang, and S.-Y. Kuo : An efficient time-based checkpointing protocol for mobile computing systems. Mobile Networks and Applications, Vol. 8, 2003, No. 6, pp. 687–697.

[25] N. Nuno : Coordinated checkpointing without direct coordination. In Proceedings of the 3rd IEEE International Computer Performance and Dependability Symposium. Pages: 23–31, 1998.

[26] R. Parameswaran, G.-S. Kang : Use of common time base for checkpointing and rollback recovery in a distributed system. IEEE Transactions on Software Engineering, Vol. 19, 1993, No. 6, pp. 571–583.

[27] Z. Tong, R.Y. Kain, W.T. Tsai : A low overhead checkpointing and rollback recovery scheme for distributed systems. In Proceedings of the 8th Symposium on Reliable Distributed Systems (SRDS'89), 10–12 October 1989, Seattle, Washington, USA. Pages 12–20.

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

[29] F. Mattern : Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. Journal of Parallel and Distributed Computing, Vol. 18, 1993, No. 4, pp. 423–434.

[30] A. Tchana, L. Broto, D. Hagimont : Approaches to cloud computing fault tolerance. International Conference on Computer, Information and Telecommunication Systems (CITS'2012), 2012; Pages: 1–6.

[31] S. Yi, A. Andrzejak, D. Kondo : Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. IEEE Transactions Services Computing, Vol. 5, 2012, No. 4, pp. 512–524.

[32] P. Kanmani, R. Anitha, R. Ganesan : A token ring minimum process checkpointing algorithm for distributed mobile computing system. International Journal of Computer Science and Network Security (IJCSNS), Vol. 10, 2010, No. 7, pp. 162–166.

[33] R-N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, R. Buyya : CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Journal Software?Practice & Experience, Vol. 41, 2011, No. 1, pp. 23–50.

[34] E. Caron, F. Desprez, D. Loureiro, A. Muresan : Cloud computing resource management through a grid middleware: A case study with DIET and Eucalyptus. In IEEE International Conference on Cloud Computing (CLOUD'09), 2009; Pages: 151–154.