DOI 10.12694/scpe.v15i2.981
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 15, Number 2, pp. 153–168. http://www.scpe.org

ADAPTIVE TIME-BASED COORDINATED CHECKPOINTING FOR CLOUD COMPUTING WORKFLOWS

BAKHTA MEROUFEL[1] AND GHALEM BELALEM[2]

Abstract. Cloud computing is a new benchmark towards enterprise application development that can facilitate the execution of workflows in business process management system. The workflow technology can manage the business processes efficiently satisfying the requirements of modern enterprises. Besides the scheduling, the fault tolerance is a very important issue in the workflow management. In this paper, we analyse and compare between some existing checkpointing strategies, then we propose a lightweight checkpointing adequate to the cloud computing and the workflows characteristics. The proposed strategy is an Adaptive Time based Coordinated Checkpointing *ATCCp*, it ensures a strong consistency without any synchronization. *ATCCp* uses the concept of soft checkpointing to minimize the storage time and it uses the *VIOLIN* topology to improve the checkpointing performances. According to the experimental results, our approach decreases the overhead and the SLA violations.

Key words: Cloud computing, Workflow, Fault tolerance, Virtual machines, Checkpointing, Coordination, *VIOLIN*, Consistency state, Stable memory, Overhead.

AMS subject classifications. 68M14, 68M15, 68M20, 68W15, 94C12

1. Introduction.

The cloud provisions pools of computing resources as services via the internet using a pay-as-you-go price model that eliminates initial costly capital investments in hardware and infrastructure. Research and academic communities can leverage the benefit of the cloud price model for their computation-intensive applications that traditionally run in HPC environments [34], [35], [36], [37] such as Amazon Web Services' HPC offering [38] or science cloud initiatives [39].

The HPC cloud market paradigm are usually smaller scale HPC users, such as small companies and research groups who have limited access to supercomputer resources and varying demand over time. The perspective of cloud providers is expanding their offerings to cover the aggregate of these HPC. The user jobs (HPC or not) are presented as a workflows to facilitate the resources and the tasks management.

The workflow is a set of atomic tasks, interconnected in a directed acyclic graph through control flow and data flow dependencies. Moving workflows to a cloud computing environment enables the utilization of various cloud services to facilitate workflow execution. Besides the workflow scheduling, the failures are an important issue to deal with during the management of workflow execution.

To ensure the cloud reliability, the system must contain a fault tolerance module to allow the cloud to continue providing the services despite the occurrence of faults. Several fault tolerance techniques are proposed such as:

- Check pointing/Restart: when a task fails, it is allowed to be restarted from the recently checked pointed state rather than from the beginning [23], [12].
- Replication: it is a process of maintaining different copies of a data item or object. Various task replicas are run on different resources, and the execution is successful if at least one replicated task is not crashed [7], [14], [40]. Replication is a resource demanding.
- Resubmission: whenever a failed task is detected, it is resubmitted either to the same or to a different resource [16].

Replication suffers from large resource consumption and resubmission can significantly delay the overall completion time in case of multiple repeating failures. In this paper we use the fault tolerance based on checkpointing. In a distributed system, since the nodes in the system do not share memory, a global state of the system is defined as a set of local states, one from each node. In this case, transit and orphan messages need to be handled; otherwise they can lead to inconsistent checkpoints [12].

[1]Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran - Es Senia, Oran, Algeria (bakhtasba@gmail.com).
[2]Dept. of Computer Science, Faculty of Exact and Applied Sciences, University of Oran - Es Senia, Oran, Algeria (ghalem1dz@gmail.com).

Orphan messages are messages whose "receive events" are part of the destination process checkpoint but the corresponding "send events" are lost. In case of a recovery, the destination process would receive those messages twice, which could result in unpredictable application behavior. On the other hand, transit messages occur when the "send events" are part of the sender-side checkpoint but the "receiving events" are lost.

Figure 1.1 presents three virtual machines (VMs), the horizontal lines represent execution process of each VM, with time progressing from left to right. An arrow from one VM to another represents a message being sent, with the send event at the base of the arrow and the corresponding receive event at the head of the arrow. Internal events have no arrows associated with them. Given this graphical representation, it is easy to verify the communication among VMs. The black boxes illustrate the checkpointing of each VM. C1 is the recorded global state and it consists of the set of VMs checkpoints.

In the scenario presented in Figure 1.1, the message *m2* is a regular message since its send and receive events are not stored in C1 (*m2* was sent and received after C1). The *m1* is a transit message because it was sent before C1 and received after C1 (the "send event" of *m1* is recorded in the global state C1 but the "receive event" is not). In case of a rollback, the VM_2 will wait the *m1* to be sent from VM_1, but VM_1 has already sent this message (according to C1). The message *m3* is an orphan message because its "receive event" is recorded in C1 but the "send event" in not recorded. In case of rollback, VM_3 receives *m3* twice (one recorded in C1 and other sent by VM_2 after the rollback) which can lead to incorrect and inconsistent computations.

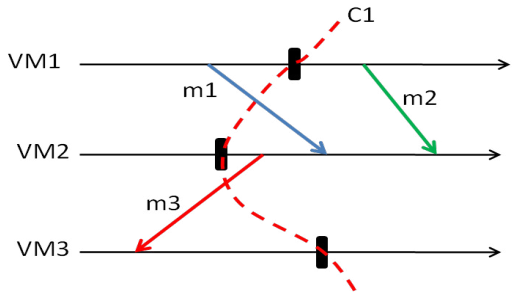

FIG. 1.1. *Communication and checkpointing among three VMs*

The checkpointing must create a consistent state. A state without any orphan message is consistent. If a consistent checkpointing do not create any transit messages then the consistency is strong. To satisfy that consistency, literature offers three main solutions:

- 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 independent checkpointing stores all the checkpoint files during the job life.
- 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 stores only one permanent checkpoint in the stable memory and it is domino-effect free.
- 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.

Even if the checkpointing is widely used in cloud computing environment [18], there are very few papers that study or take into consideration the influence of checkpointing technique. The random choice of checkpointing technique adds large overhead and decreases the system performance considerably. After studying a real execution trace of a cloud computing, it is proved that 80% of communications happen inside the server (among VMs) [8].

In this paper we propose an intra-sever checkpointing strategy. The inter-server can easily managed by using the communication induced checkpointing at servers level [6]. The proposed checkpointing approach is low-

overhead time-based coordinated checkpointing (*ATCCp*). *ATCCp* technique does not need any synchronization and it stores only one checkpointing file in the stable memory which decreases the overhead and the resource consumption. An initiator is selected in *ATCCp* to manage the resynchronization and to solve the timer problems.

The rest of this paper is structured as follows: Section 2 explains some related works. Section 3 presents the adopted system model and its characteristics. Section 4 analysis in detail the time based coordinated checkpointing strategies proposed in literature. In section 5, a description of *ATCCp* is given; a comparative study is also presented in the same section. The experimental results are presented in section 6. The conclusion and future works are presented in section 7.

2. Related works. Besides the scheduling, fault tolerance is a very important issue in workflow management. Literature proposes many works in this field. The authors in [7], [14] use the replication to tolerate the failures. This strategy is very effective when the system uses a deadline for the running application. However, the replication is resource demanding and it can add complexity to the system. The resubmission is also another strategy to ensure the correct execution of the workflow. The authors in [16] balance between the resubmission and the replication to minimize the disadvantages of each strategy. Even with this combination, fault tolerance decreases always the system performance and increases the SLA (service level agreement) violation [1].

The previous strategies (replication and resubmission) are used only for fault tolerance. Checkpointing is a fault tolerance technique but it can be used to ensure or improve other services. In [1], [15], [13] the authors use the checkpointing to ensure a fault tolerant scientific workflows. In this case, the system records periodically its state by checkpointing all the running tasks. The proposed approach in [1] uses two phase blocking coordinated checkpointing which considerably increases the system overhead. The authors in papers [15], [28] use independent checkpointing which causes the domino effect. Independent checkpointing is used also in [4], [5] to improve the migration and ensure the load balancing in cloud servers.

The fault tolerance service proposed in [28] and [29] chooses between replication and checkpointing to tolerate the workflow depending on several criteria such as the cost and the overhead. In [27], a comparison is done between the replication only and the combination between replication and checkpointing. The experimental results prove that the checkpointing improves the replication performance.

It is clear that the checkpointing is a very popular technique and it can be used to ensure many services at the same time. However, random choice of checkpointing technique can increase the system overhead and the SLA violation. According to Section 1, coordinated checkpointing ensures the consistency without the scalability and it is the contrary in case of uncoordinated checkpointing.

Time based coordinated checkpointing is a combination between both strategies (coordinated/ uncoordinated) and it ensures both scalability and consistency with the minimum overhead. Many papers studied and used this type of checkpointing. The majority of the proposed time based checkpointing suffer from many problems and imperfections. Before presenting and analyzing this type of checkpointing, we present in the next section the system model.

3. System model. The cloud computing is a set of datacenters geographically distributed. Each datacenter contains multiple physical machines (servers/Hosts). The hypervisor [9] in each server provides a virtualized hardware environment to support running multiple operating systems concurrently in different virtual machines (VMs) using one physical server (See Figure 3.1). The VM can accept a special user request to execute an application. All the virtual machines can concurrently serve, with different operating systems and software configuration environments according to the need of the user. The kernel level in the VM contains the communication and the task management modules.

We have extended this architecture by implementing a checkpointing module (CP module) inside each VM. We assume also that each datacenter contains a stable storage memory where checkpoint files will be stored. All the servers are connected to the stable memory of their datacenter. The broker is the customer representative in the cloud environment. It allocates resources for applications/services on multiple VMs in order to fulfill requests of the client, it negotiates also with the resource provider (the cloud computing) to obtain the most cost-effective resources. The contract between the broker and the resource provider is represented by the service level agreement (*SLA*). The client, the broker and the cloud datacenters are connected by a network.

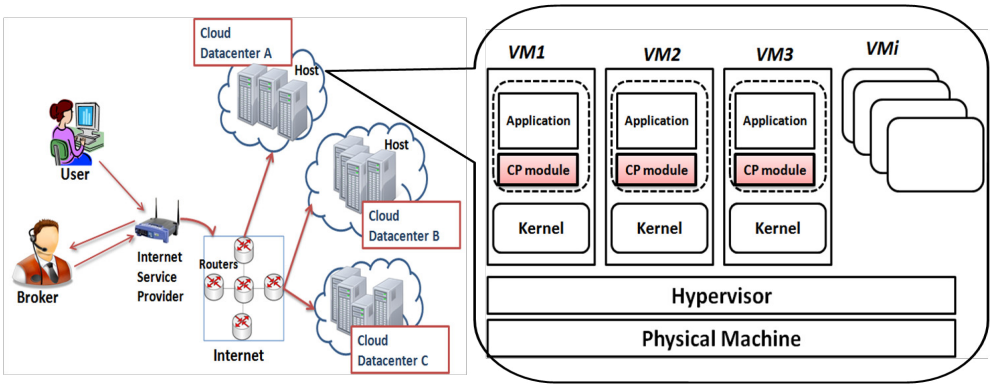

FIG. 3.1. *System model*

The application models in cloud computing can vary from multi-tier web applications such as e-commerce to scientific applications (parallel and data-driven applications and workflows) [15]. Typically such applications consist of several tasks, which communicate with each other. A task consists of some computation and communication phases. Communications among tasks create dependencies [15], [19]. The workflow is used to indicate the temporal relationship among tasks and to present their dependencies. In order to show how the workflow is presented in the system, we used an oriented graph as shown in Figure 3.2.

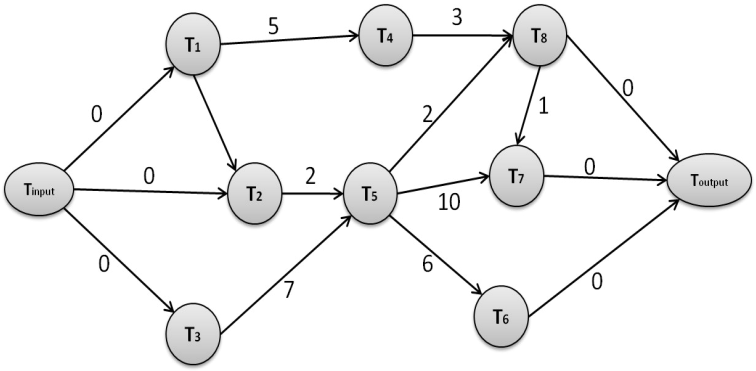

FIG. 3.2. *Workflow example with eight tasks*

The oriented graph consists of eight tasks from T_1 to T_8, and two dummy tasks: t_{input} and t_{output}. The number above each arc (directed edges) shows the estimated dependency factor between the corresponding tasks. In other words, if task T_1 sends five messages to task T_4 then the $ArcWeight = 5$. The average communication rate μ is calculated by the formula 3.1.

$$\mu = \frac{\sum_{l=0}^{r} ArcWeight_l}{r} \tag{3.1}$$

where r is the number of arcs in the workflow. The number of tasks executed in parallel depends on the number of available VMs.

Since the tasks are dependent to each other by their communications, transit and orphan messages can exist during the creation of checkpoints which makes the independent checkpointing inadequate to ensure a consistent state. However, coordinated checkpointing is very expensive in term of overhead and energy consumption. To resolve this problem, we used an adaptive time based coordinated checkpointing *ATCCp*.

The work in [6] is similar to our proposition but it assumes that VMs clocks are perfectly synchronized, which may not be true in real system. The work in [6] ensures also only a simple consistency without taking into consideration the transit messages. Our *ATCCp* deals with both transit and orphan messages to ensure a strong consistency with the minimum overhead and without any need for clock synchronization.

4. Time based coordinated checkpointing. Time based coordinated checkpointing is a combination between coordinated and uncoordinated checkpointing, it is proposed to avoid the problems of each checkpointing strategy. To facilitate the understanding of our work, we present in Table 4.1 the principal symbols cited and used in the rest of this paper.

TABLE 4.1
Time based checkpointing parameters

Symbol	Signification
ρ	Timer drift
T	Initial timer value
MD	Maximum deviation
D	Initial deviation of checkpointing interval
ED_C	Effective deviation for the consistency
ED_R	Effective deviation for the recoverability
csn	Checkpoint sequence number
t_{max}	Maximum delivery time of a message
t_{min}	Minimum delivery time of a message
FS_i	Vector of sending flags of VM_i
T_C	Checkpointing interval

Time based coordinated checkpointing is based on the idea that if the VMs create their checkpoints at the same time (without any deviation), orphan messages will not exist [17], [18], [23]. However, the clock synchronization among all the VMs is impossible in real systems. To overcome this problem, time based coordinated checkpointing uses a timer instead of real clock and it assumes that the VMs have loosely synchronized clocks. It assumes also that all the VMs timers are approximately synchronized with a deviation from real time of some value ρ, which presents the clock drift.

If a timer is started on each of two VMs at the same time, and each VM has clock drift rate equal to ρ, the timers will expire at $2\rho T$ seconds apart, where T was the initial timer value and $10^{-8} < \rho < 10^{-5}$. Assuming this, the clocks will exhibit a maximum drift of $2\rho nT$ after N checkpoint intervals [19]. Time based coordinated checkpointing ensures a strong consistency if it satisfies two conditions: consistency condition and recoverability condition.

4.1. Consistency condition. The consistency condition ensures that the system will not create any orphan message during the checkpointing process. So the checkpointing must ensure that if the "sent event" of a message m is not recorded in the checkpoint file of the source, the "receive event" will not be recorded also in the checkpoint file of the destination [12]. To deal with orphan messages and satisfy the consistency condition, many solutions are proposed and implemented in the literature.

In [6], [17], [25], the authors assume that the timers (real or logical) are perfectly synchronized. In this case, all the VMs will create their checkpoints at the same time. In Case 1 of Figure 4.1, VM_0 and VM_1 create their checkpoint $C_{0,0}$ and $C_{1,0}$ at the same time (in $C_{i,j}$: the i is the VM identifier and j is the checkpoint csn). If VM_0 sends a message m to VM_1 after the checkpointing, it is sure that the message will be received after the checkpoint of VM_1, so m will never be an orphan message. However, the perfect timer synchronization among VMs is not possible in real systems.

To avoid the synchronization condition, the papers [18], [19], [24] take into account the timer deviation ρ. In this case, orphan message can be created. The second case of Figure 4.1 illustrates a possible situation where VM_0 creates its checkpoint $C_{0,1}$ at time t_1 and then sends a message *m1* to VM_1. After receiving *m1*, VM_1 creates the checkpoint $C_{1,1}$. So *m1* will be an orphan message and the consistency is not assured. If VM_0 did not send *m1* during the timer deviation (until VM_1 creates its $C_{1,1}$), the consistency will be assured.

Based on this observation, the papers [18], [19], [24] propose to block the message sent during a certain period after the checkpointing. Using time based checkpointing assumptions, it is possible to specify approximately

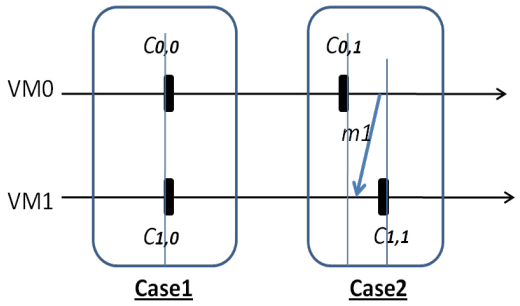

FIG. 4.1. *Possible cases of consistency condition*

the time interval after the checkpointing where potential orphan messages can be created. If $\rho \neq 0$ then the Maximum Deviation can be calculated by Formula 4.1:

$$MD = D + 2\rho nT \tag{4.1}$$

where D is the initial deviation of checkpointing intervals. A message sent during MD after the checkpointing can cause an inconsistency. The MD period can be minimized by taking into account the transmission time of the message, so the effective deviation of consistency ED_c will be (See Formula 4.2):

$$ED_c = MD - t_{min} \tag{4.2}$$

where t_{min} is the minimum delivery time of a message inside the server. If $MD = 0$ then the timers are fully synchronized (case 1 of Figure 4.1). Preventing a VM to send messages during ED_c period (by buffering this message until the end of ED_c) can resolve the problem. However, blocking the communication increases the response time and the SLA violation.

Both the timer synchronization and the blocked communication decrease the system performances. To ensure the consistency, another solution proposes the use of piggybacked messages [20], [21], [22], [23]. In this case, each message will piggyback some extra data and the receiver will decide to create or not the checkpoint according to this data. One of the piggybacked data is the checkpoint sequence number (*csn*) of the local VM. The *csn* is an integer number that will be incremented each time the VM creates its checkpoint file. In case of time based coordinated checkpoints, the *csn* in all VMs of the server must be the same since all these VMs create their checkpoints using the same checkpointing interval T_C.

So, if a VM receives a message where the piggybacked *csn* of the message is greater than the local *csn*, the receiver VM knows that the sender has already created its checkpoint before sending the message. In this case and to avoid the creation of orphan message, the VM receiver must create its checkpoint before dealing with the received message (it buffers the message, creates the checkpoint and then computes this message).

In case 2 of Figure 4.1, the message *m1* will piggyback the $csn = 1$, when VM_1 receives this message, it compares the local $csn = 0$ with the *csn* of the message, since the local *csn* is less than the sent *csn*, VM_1 will know that VM_0 has created its checkpoint before sending *m1*, so VM_1 will be forced to create the checkpoint $C_{1,1}$ before dealing with *m1*. This strategy increases the delivery time of messages.

4.2. Recoverability condition. The recoverability condition ensures that the system will not create any transit message during the checkpointing process. The checkpointing must ensure that if the "receive event" of a message m is recorded in the checkpoint file of the destination, the "send event" will also be recorded in the checkpoint file of the source [12].

Using an illustrating example in Figure 4.2, transit messages can exist even if the VMs create their checkpoints at the same time. In case 1 of Figure 4.2, both VM_0 and VM_1 created their checkpoints at the same time, VM_0 has sent *m0* before the checkpointing but VM_1 has received *m0* after its checkpoint which makes *m0* transit.

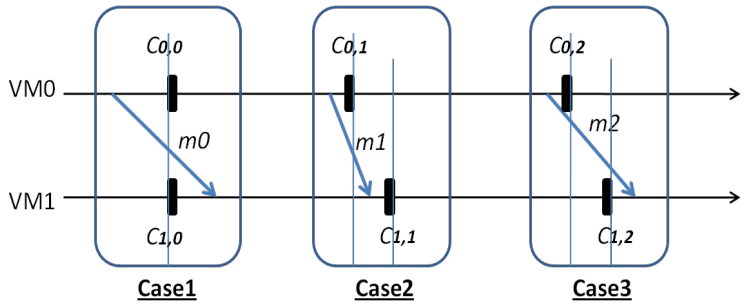

FIG. 4.2. *Possible cases of recoverability condition*

The clock deviation also does not always cause a transit message. In case 2 of Figure 4.2, the message *m1* is not transit since its send and receive events are not recorded in the global state. So the existence of transit message is related more to the transfer time of the message (time from sending a message to receiving it) compared to the checkpointing interval. The case 3 of Figure 4.2 presents the same situation of case2. However in case 3, the transfer delay of *m2* was greater than the transfer delay of *m1* in case 2, so *m2* is received after the checkpointing of VM_1 but sent before the checkpointing of VM_0 which makes it transit message.

To deal with the problem of transit messages, the paper [24] proposes to block the communication a certain period just before the checkpointing time. The transit message is a message sent before the checkpointing of a VM_i and received after the checkpointing of VM_j. So if we prevent VM_i to send any message that can be received after the checkpointing of VM_j, the problem will be resolved.

Using time based checkpointing assumptions and adopting the same reasoning of blocked communication used to resolve the problem of case2 in Figure 4.1, we can estimate the time interval that can create potential transit messages. This interval named effective deviation of recoverability ED_R is calculated by Formula 4.3:

$$ED_R = MD + t_{max} \tag{4.3}$$

where t_{max} is the maximum delivery time of a message in the server. However, blocking the communication always adds an extra overhead to the system.

To avoid blocking communications, the papers [18], [20], [23] use the send-messages logging. In this strategy, the VM logs all the messages sent. In case of rollback of the receiver, it can ask the sender to re-sent these messages. But how many messages must be logged and until when. The only period that can create potential transit message is ED_R before the checkpointing time [20], [24], [25]. So logging the message sent during this period can resolve the transit message problem. We summarized all the used techniques that ensure the consistency and the recoverability conditions in Figure 4.3.

5. The contribution. In the previous section, we cited and explained the existing time-based coordinated checkpointing works. The majority of these works deal with the consistency by blocking the communications or piggybacking the messages. There are also some works that do not consider important parameters such as: timer drift, checkpointing overhead, message transfer time. Some works ensure only a simple consistency. So each strategy suffers from some imperfections and does not deal with some problems. And here comes our contribution as a solution of these imperfections and problems.

5.1. Adaptive Time based Coordinated Checkpointing (*ATCCp*). The proposed contribution in this paper is an adaptive time based non-blocking coordinated checkpointing (*ATCCp*). *ATCCp* ensures a strong consistency without overheating the system and without synchronizing the timers or blocking the communications. *ATCCp* has three phases: Initiator selection, Checkpointing and Resynchronization.

5.1.1. Initiator selection Phase. In this phase, each server (host) selects an initiator of checkpointing. The initiator is the most powerful and less busy VM in the server. In *ATCCp*, the role of the initiator is the general resynchronization and the checkpointing control.

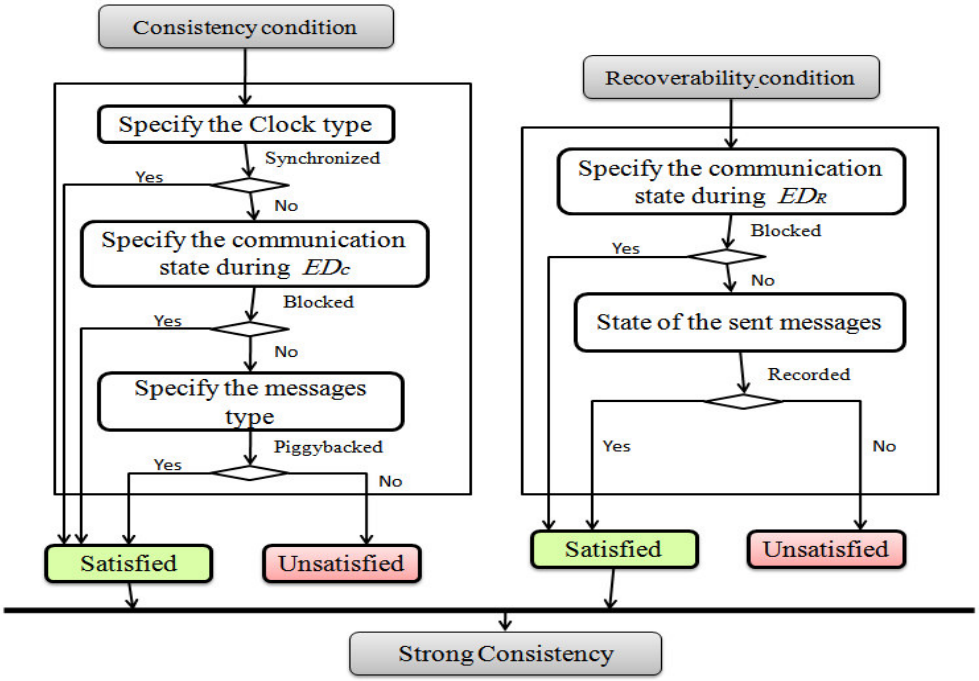

FIG. 4.3. *Time based checkpointing strategies*

5.1.2. Checkpointing Phase. Periodically, each VM creates its checkpoint independently without any control message exchange. *ATCCp* ensures a strong consistency by dealing with both consistency and recoverability conditions. Since *ATCCp* is not blocking or synchronizing checkpointing, recording and piggybacking messages are the only solutions to ensure a strong consistency. Contrary to the previous cited works, our approach ensures the desired consistency with the minimum overhead by reducing the number of recorded and piggybacked messages.

In *ATCCp*, the messages sent during ED_C after the checkpointing are potential orphan messages. To avoid blocking the communication during ED_C (such as the strategies in [18], [19], [24]), when a VM sends a message during ED_C to another VM (in this case the "send event" is not recorded), the VM receiver buffers the message and creates its checkpoint file, then it computes the message. In this case, the "receive event" will be also unrecorded in the checkpoint file of the receiver. However, not all the messages sent during ED_C will force the receivers to create their checkpoints. The VM destination is forced to create the checkpoint if the *csn* of the message (actually it is the *csn* of the source VM) is greater than the local *csn* of VM destination.

Besides the *csn*, the piggybacked message will also contain the "time to next checkpoint TNC" representing the rest of the time before the next checkpoint of the source. The goal of this information is the resynchronization. To reduce the number of piggybacked message in *ATCCp*, a VM_i piggybacks only its first application message sent during ED_C (after it has create its checkpoint) to a VM_j.

Each VM_i uses a Boolean vector FS_i of N size, where N is the number of VMs in the server. The FS_i is used to identify the first message sent to a VM destination, when VM_i sends a message to VM_j then $FS_i[j] = 1$. After each checkpointing the vector is initialized ($FS_i[j] = 0$ / $i \neq j$). The second computation message sent to the same destination can not force the checkpointing creation because the source and the destination have the same *csn*.

In *ATCCp*, the potential transit messages in VM_i are those sent during the period ED_R. So only the messages sent during this period will be stored in the checkpoint file and VM_i continues its execution without blocking its communications. In case of rollback, the transit message will be detected and the source will send it again to the destination. However in paper [20], the author has proved that if we use the same checkpointing interval (even with deviations) and if $T_c + MD > t_{max}$ (the maximum difference between the sender *csn* and

receiver *csn* is 1) then it is sufficient to store only the messages sent during the last checkpointing interval.

5.1.3. Resynchronization Phase. In time based checkpointing approachs, the clock deviation increases with $2\rho nT$ after n checkpoint intervals [18], [19]. The goal of the resynchronization phase is to reinitialize the value of ρ. If $\rho = 0$ then the Maximum deviation will be: $MD = D + 2\rho nT = D$ where D is the initial deviation of checkpointing intervals. The synchronization can be executed by the initiator or the ordinary VM.

- The initiator can start the resynchronization when the deviation MD exceeds a certain threshold or the size of logged messages exceeds the buffer size of VMs. In this case, the initiator sends special messages to VMs to synchronize their timers.
- An ordinary VM can use its piggybacked message to minimize the deviation between the local timer and the destination timer. So when the VM receives the piggybacked message, it uses the *csn* to decide to create or not the checkpoints and it uses the synchronization data (*time to next checkpoint TNC* and the message transfer delay) to minimize the clock deviation.

The combination between the initiator and the VM resynchronization decreases considerably the overhead. The initiator synchronization overheads the server by control messages but it decreases the resource consumption, i.e. the number of possible orphan and transit messages will be reduced since the ED_R and ED_C are reduced. The VM synchronization adds more data on the piggybacked messages but it decreases the initiator resynchronization rate.

However, if the timer of the sender is faulty, the erroneous timer information will be spread to the receiver. Besides, since the transmission delay between the sender and the receiver is variable, the timer information from the sender may not reflect the correct situation when the message finally arrives at the receiver. To achieve more accurate timer synchronization, we can utilize the timers in the initiator as an absolute reference. The Figure 5.1 illustrates a part of *ATCCp* process (only the consistency condition and the resynchronization of an ordinary VM) in two communicating VMs (VM_i and VM_j). The green arrow presents a communication message sent from VM_i to VM_j.

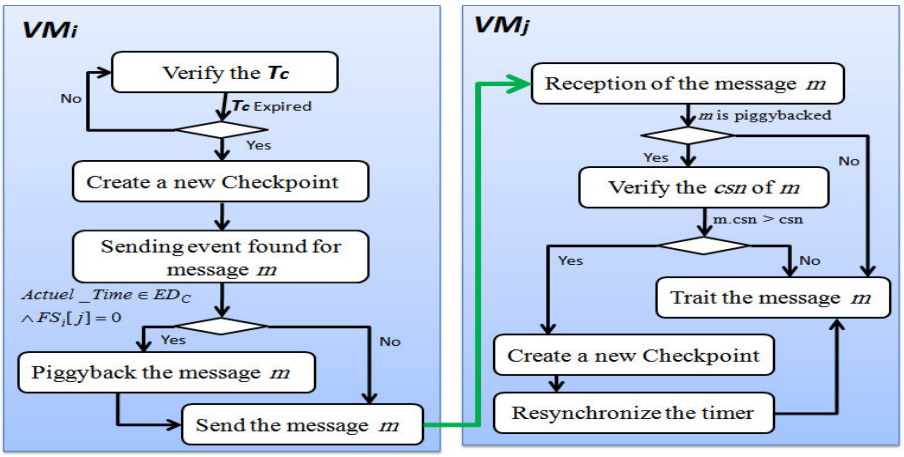

FIG. 5.1. *ATCCp process in two VMs*

Besides the checkpointing technique, it is proven that 70% of checkpointing overhead is caused by the storage phase (called the checkpointing latency), i.e. the time needed by the VM to store its checkpoint file in the stable memory. During the storage phase, the VM suspends its running task to execute the storage which means that the task will be delayed and therefore the whole workflow will be delayed also.

To minimize the storage time, we used soft checkpointing. When the VM creates its checkpoint file, it sends it to the initiator (the file) and continues immediately the running task. When the initiator receives all the checkpoints files, it sends these files to the stable memory. The initiator can use I/O strategies such as data sieving and collective I/O [26] to improve the storage time.

Since *ATCCp* is a server checkpointing technique, we used an enhanced version of *VIOLIN* topology [2]. The *VIOLIN* (virtual networked environment) consists of multiple VMs connected by a virtual network. VMs of the same server are connected by *VIOLIN* switches running in each server. In the classical *VIOLIN* proposed in [2], the checkpoints are taken by *VIOLIN* switches from outside VMs using the strategy proposed in [3]. Although communications pass through the *VIOLIN* switch, each VM is responsible to manage its sent/received messages. *VIOLIN* topology is destined for the fault tolerance, but it is very sensitive to the *VIOLIN* switch failure. If a virtual switch in *VIOLIN* fails (in case of in-transient failures), the checkpointing will be impossible and the cloud will lose its reliability.

To address this problem in *VIOLIN* topology we combined between the host topology of Figure 3.1 and the classical *VIOLIN* by implementing a checkpointing module in each VM. One VM will be selected as *VIOLIN* switch and in case of failure; another VM can replace it immediately. *ATCCp* does not need any extra-control messages or extra-resource, so its implementation will not need major modifications.

To facilitate the checkpointing process, we suppose that the *VIOLIN* switch is always the checkpointing initiator because it has a global view about the existing VMs in the host (server). In case of any need of control (for example: specifying the maximum deviation time or starting the resynchronization), the *VIOLIN* switch can forward the control messages without losing time in collecting data about the actual situation. Besides that, the *VIOLIN* switch uses the server clock to solve the timer problem.

5.2. Analysis and comparative study. To analyze our contribution, we used the example of Figure 5.2.

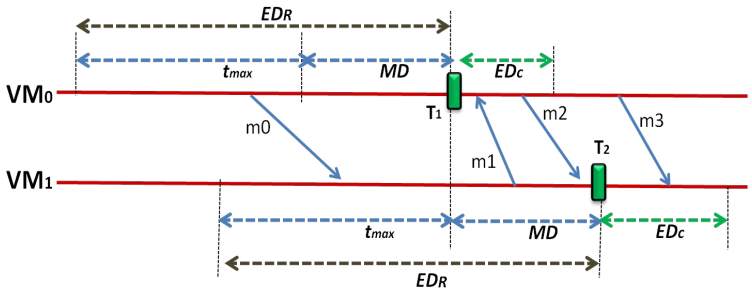

FIG. 5.2. *ATCCp parameters with an example*

Figure 5.2 illustrates all the time intervals used in time based coordinated checkpointing (see Table 4.1). It presents also an example of the checkpointing process in two VMs .

- Each VM estimates its ED_R and ED_C.
- VM_0 and VM_1 create their checkpoint files (*T1* and *T2* respectively) independently
- Each VM stores the messages sent during its ED_R in the checkpointing file. Those messages are probably transits. In case of rollback, the transit message will be sent again to its destination. In Figure 5.2, the message *m1* is transit and it will be stored in the checkpoint file of VM_1. In case of failure and rollback, the VM_0 requests VM_1 to resend *m1*, and *m1* is already stored in the checkpointing file of VM_1.
- The messages sent during ED_C of the source can be an orphan message, so if the message is sent for the first time to a destination, that message will be piggybacked by some data (specially the *csn*) and it will force the destination to create its checkpoint before the checkpointing (only if *csn* of the source is greater than the *csn* of the destination). In this case, the message will be regular and the *csn* of the source and the destination will be the same. In Figure 5.2, the VM_0 sent *m2* to the VM_1 during its ED_C and since this message is the first message sent to VM_1 after the last checkpoint T1, it will be piggybacked by some data. At the reception of $m2(Source_{csn} = 2 > Destination_{csn} = 1)$, the VM_1 creates its checkpoint T2 before treating this message (not illustrated in the Figure 5.2) and it will also resynchronize its timer. If VM_0 sends a second message to VM_1 during ED_C, the message will not be piggybacked with extra-data and it will not force the destination to create its checkpoint file since *csn* of VM_0 and *csn* of VM_1 are equal.

Our approach solves many problems of time based coordinated checkpoints proposed in literature and it does not omit the majority of parameters. Table 5.1 presents a comparison between our *ATCCp* and the other time based coordinated checkpointing cited in this paper.

TABLE 5.1
Comparison between time based checkpointing

Paper	ρ	**Transfer time**	**Timer prob**	**Blocking**	**Sys type**	**CP type**	**Synch**	**Recov**	**Piggy-back**
[17]	No	No	No	Yes	Dist	H	SM	No	No
[18]	Yes	Yes	No	Yes	Dist	H	SM	Yes	No
[19]	Yes	Yes	No	Yes	Dist	H	SM	Yes	No
[20]	Yes	No	No	No	Mob	S/H	SM	Yes	Yes
[21]	Yes	Yes	Yes	No	Mob	S/H	CM	Yes	Yes
[22]	Yes	No	No	No	Mob	H	CM	No	Yes
[23]	Yes	No	No	No	Mob	S/H	CM	Yes	Yes
[24]	Yes	Yes	No	Yes	Mob	H	CPM	Yes	No
[25]	No	Yes	No	No	Dist	H	SM	Yes	Yes
[6]	No	No	No	No	Cloud	H	CM	No	Yes
ATCCp	Yes	Yes	Yes	No	Cloud	S/H	CM/ SM	No	Not always

The comparison criteria are: considering the timer drift (ρ), considering the transfer time of messages (*Transfer time*), dealing with timer problems (*Timer prob*), if the system freezes communications or not (*Blocking*), the system type where the checkpointing strategy was implemented for (*Sys type*: Mob (Mobile), Dist(Distributed) or Cloud), the resynchronization tools (*Synch* represents the type of message where the synchronization is piggybacked: SM (Special messages), CM (Computing messages) or CPM (Checkpointing messages)), ensuring the recoverability condition (*Recov*), using the piggybacked messages (*Piggyback*). And finally the type of checkpoint files: Hard (*H*) or Soft (*S*). In hard checkpointing, the VM stores its state in the stable memory directly. In the soft type, the VM stores its checkpoint in the local memory or sends it to a storage manager and continues its execution immediately. The S/H means that the checkpointing uses both soft and hard checkpoint files.

6. Performance evaluation. In this section, we evaluate the performances of our checkpointing scheme using *CloudSim* simulator [11]. The *CloudSim* is currently the most sophisticated discrete event simulator for Clouds; it is an open source, scalable and low simulation overhead simulator [11]. It is used in many papers to simulate the scientific workflows [31], [32], [33].

The experimentations in [32] prove that *CloudSim* has the best accuracy in workflows performances. The accuracy is defined as the ratio between the predicted performances using a specified metric and the real performances.

We did not compare our *ATCCp* with the other time based coordinated checkpoints because the *ATCCp* is a solution of classical time based checkpointing problems, which makes it clearly better than those strategies. However, we studied the *ATCCp* behavior compared to coordinated checkpointing *CCp* used in [1], [16], [35] and independent checkpointing *ICp* used in [15], [28], [36]. The goal is proofing that our approach ensures a strong consistency with the minimum cost (like the *CCp*) and with the minimum overhead (like the *ICp*). Simulation parameters are presented in Table 6.1.

6.1. Makespan vs Number of VMs. In the first experiment, we measured the impact of number of checkpoints on the makespan (scalability) for the three checkpointing strategies *ATCCp*, *ICp* and *CCp*. We used the parameters cited in Table 6.1, except that the cloudlet length is fixed at 1000 MIPS, μ=50, the number of VMs is 40 and the checkpointing interval: $100 < T_C < 600s$. The makespan is measured by Formula 6.1.

$$makespan = max_{T_i \in T} ExectTime(T_i) \tag{6.1}$$

where T_i is the task i in the running workflow T. And $ExecTime$ is the execution time of that task (Task i).

TABLE 6.1
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 μ	2-100
Checkpoint interval$CP_{Interval}$	100-500 second
Failure rate λ	2 to 5 per period

The results are presented in Figure 6.1. Incrementing the checkpointing number increases the makespan in all the strategies (*ATCCp*, *CCp*, *ICp*). *ATCCp* uses soft checkpointing and it does not need any coordination except in case of resynchronization, which makes its performances similar to the *ICp* with an average difference of 10%.

In our work, we present the $SLA_{Violation}$ by the overhead caused during the checkpointing [8]. The *SLA* is a contract between customers and service providers of the level of service to be provided [1]. So the customer can precisely specify the margin for acceptable overhead (20% for example). If the overhead caused by the checkpoint exceeds the margin, then a SLA violation will be detected. In this experiment, $SLA_{Violation}$(ATCCp)=13%, $SLA_{Violation}$ (CCp)=27% and $SLA_{Violation}$ (ICp)=11.78%.

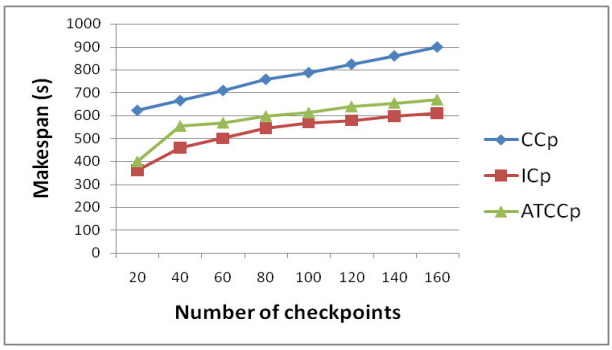

FIG. 6.1. *Makespan vs Number of checkpoints*

6.2. Cost vs Number of checkpoints. As the cost is a very important parameter in cloud environment, we used the parameters of the first experiment to analyze the cost of all checkpointing strategies. The total cost is calculated using Formula 4.3:

$$TotalCost = \sum_{i=1}^{k} Cost(T_i) + \sum_{j=1}^{m} Cost(CP_j) \tag{6.2}$$

where k is the number of tasks in the workflow and m is the number of checkpoints created during the workflow execution. $Cost(T_i)$ is the cost of running the task T_i in the specified resource, it includes the communication cost (running the send/receive events). The cost of the j^{th} round of checkpointing is $Cost(CP_j)$ and it includes the coordination cost, the state recording cost and the storage cost.

According to the results illustrated in Figure 6.2, the cost of *CCp* is less than the cost of *CCp* and *ICp* (specially in case of high checkpointing rate). The only checkpointing communication needed in *ATCCp* is during the resynchronization so the coordination cost will be reduced. Also the use of VMs resynchronization minimizes the timer drift, so the periods ED_C and ED_R will also be reduced. In this case, the piggybacked messages and the number of transit messages stored in checkpoint files will be reduced, which means less storage

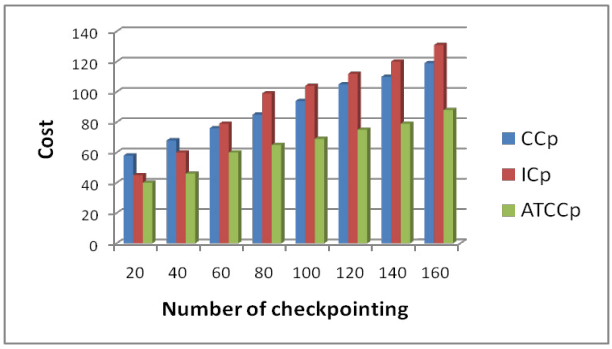

FIG. 6.2. *Cost vs Number of checkpoints*

cost and less bandwidth consumption. Also in *ATCCp*, only one checkpointing state is needed to ensure a correct rollback.

In *CCp*, the server must record all the sent messages to avoid the transit messages (ensuring the recoverability condition) and the initiator must coordinate with the VMs in each checkpointing round, which increases the cost. The cost of *ICp* was significantly increased when the checkpointing rate increased because this type of checkpointing needs to store all the checkpoint files to ensure the correct rollback (domino effect). So the *ICp* cost is due to the checkpointing latency (storage).

6.3. Communication rate vs cost. Communication rate μ represents the dependency degree between tasks of the workflow. This dependency causes a major problem in checkpointing techniques. In this experiment, we measured the cost of the *ATCCp* and *CCp* checkpointing in the system with different μ. The *ICp* is not affected by the communication rate since no coordination is needed.

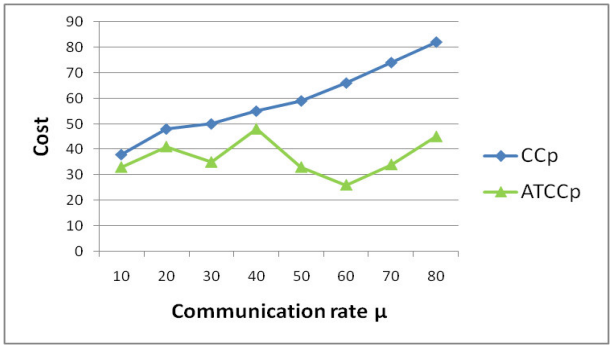

FIG. 6.3. *Communication rate Vs cost*

The results (See Figure 6.3) prove that increasing the communication rate can improve the *ATCCp* performances since the communication among VMs (specially during the ED_C) decreases the timer drift (using the piggybacked messages). So the period ED_R will be reduced and the initiator resynchronization frequency will be decreased. In *CCp*, a high μ implies more storage space needed to ensure the recoverability condition. The number of piggybacked messages in *CCp* will increase since this checkpointing does not block the communications. In our *ATCCp*, piggybacked messages are sent only during ED_C (See Section 5).

VIOLIN topology has improved both checkpointing techniques with almost 14.3% since the initiator is *VIOLIN* switch (it has a global view about the communication among VMs), so no time is needed to select an initiator, or to collect data about communications or VMs states in the server.

6.4. Rollback duration Vs failure rate. In this experiment, we fixed all the parameters cited in Table 6.1, but the failure rate λ is varied from 3 to 24 per period. Our goal is to measure the time needed for the

rollback in case of failure. The performances of *CCp* and *ATCCp* are approximately the same because both strategies need only the last stored checkpoint file to resume the work. However, *ICp* has to ensure the correct state among all the stored checkpointing files, which can lead to the domino effect and therefore the rollback duration will increase (See Figure 6.4).

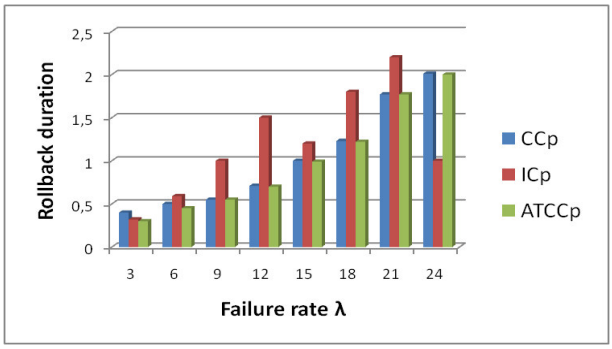

FIG. 6.4. *Rollback duration Vs failure rate*

7. Conclusion. When computationally-intensive workflows are executed, fault handling is very important, since the failure of a single component might lead to an abandonment of the entire workflow. This may lead to the loss of the stability and the reliability in the system. Since both the customer and the cloud computing are bound by the SLA contract, the choice of the checkpointing technique is very critical.

In this paper we proposed a fault tolerance service to tolerate failures using checkpointing technique. Our approach is a time-based coordinated checkpointing *ATCCp*. It ensures a strong consistency with the minimum control messages and without blocking the VMs communications. *ATCCp* uses the *VIOLIN* topology and the soft checkpoint to improve the checkpointing performances.

To evaluate our approach, we compared it with the most popular checkpointing techniques: coordinated and uncoordinated checkpointing. The experimental results prove that *ATCCp* omits the problems of classical approaches. It decreases the overhead, minimizes the SLA violation and ensures a rapid rollback. Our work focuses on intra-server checkpointing. However, *ATCCp* can be easily extended to support the inter-server checkpointing by using multi-level checkpointing (Communication induced checkpointing at servers level and *ATCCp* at VMs level). In the future work, we will use a real platform such as eucalyptus [30] to implement the *ATCCp*.

REFERENCES

[1] M. Zhang, H. Jin, X. Shi and S. Wu, *VirtCFT: A Transparent VM–Level Fault-Tolerant System for Virtual Clusters*, IEEE Proceeding of the 16th International Conference on Parallel and Distributed Systems (ICPADS), 8–10 Dec. 2010, Shanghai, pp. 147–154.

[2] A. Kangarlou, P. Eugster and D. Xu, *VNsnap: Taking Snapshots of Virtual Networked Infrastructures in the Cloud*, IEEE Transactions on Services Computing, 5(4): 484–496, 2012.

[3] F. Mattern , *Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation*, Journal of Parallel and Distributed Computing, 18(4): 423–434, 1993.

[4] S. Di ,Y. Robert, F. Vivien, D. Kondo, C–L. Wang and F. Cappello, *Optimization of cloud task processing with checkpoint–restart mechanism*, International Conference for High Performance Computing, Networking, Storage and Applications (SC2013), Nov. 17–22, 2013, Dever, Colorado, USA.

[5] D. Nguyen and N. Thoai, *EBC: Application–level migration on multi–site cloud*, The International Conference on Systems and Informatics (ICSAI), 19–20 May, 2012, pp. 876–889.

[6] 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, 6(2): 61–71, 2013.

[7] O. Ben–Yehuda, A. Schuster, A. Sharov, M. Silberstein and A. Iosup, *ExPERT: Pareto–Effcient Task Replication on Grids and a Cloud*, IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), 21–25 May, 2013, pp. 167–178.

[8] R. Jhawar, V. Piuri and M. Santambrogio, *Fault Tolerance Management in Cloud computing : A System–Level Perspective*, IEEE Systems Journal, 7(2): 288–297, 2013.

[9] M.N.O. Sadiku, S.M. Musa and O.D. Momoh, *Cloud computing: Opportunities and challenges*, Potentials IEEE Journal, 33(1): 34–35, 2014.

[10] W. Zhao, Y. Peng, F. Xie and Z. Dai, *Modeling and Simulation of Cloud Computing: A Review*, IEEE Asia Pacific Cloud Computing Congress (APCloudCC), 14–17 Nov. 2012, Shenzhen, pp. 20–24.

[11] R. Calheiros, R. Ranjan, A. Beloglazov1, C. De Rose and R. Buyya, *CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms*, Software: Practice and Experience journal, 41(1): 23–50, 2011.

[12] G. Cao and M. Singhal, *On coordinated checkpointing in distributed systems*, IEEE Transactions on Parallel and Distributed Systems, 9(2): 1213–1225, 1998.

[13] I. Egwutuoha, S. Chen, D.Levy, B. Selic and R. Calvo, *A Proactive Fault Tolerance Approach to High Performance Computing (HPC) in the Cloud*, Second International Conference on Cloud and Green Computing (CGC2012), 1–3 Nov. 2012, Xiangtan, pp. 268–273.

[14] L. Arockiam and G. Francis, *FTM–A Middle Layer Architecture for Fault Tolerance in Cloud Computing*, IJCA Special Issue on Issues and Challenges in Networking, Intelligence and Computing Technologies, ICNIT 2012, No.2, pp. 12–16.

[15] T. Nguyen and J-A. Desideri, *Resilience Issues for Application Workflows on Clouds*, In Proceeding of the 8th International Conference on Networking and Services (ICNS2012), Mach 2012, Sint–Maarten (NL).

[16] P. Palaniammal and R. Santhosh, *Failure Prediction for Scalable Checkpoints in Scientific Workflows Using Replication and Resubmission task in Cloud Computing*, International Journal of Science, Engineering and Technology Research (IJSETR), 2(4): 985–991, 2013.

[17] S. Neogy, A. Sinha and P. K. Das, *Checkpointing with synchronized clocks in distributed systems*, International Journal of UbiComp, 1(2): 64–91, 2010.

[18] N. Neves and W.K. Fuchs, *Coordinated checkpointing without direct coordination*, IEEE International Computer Performance and Dependability Symposium (IPDS98), 7–9 Sept. 1998, Durham, NC, pp. 23–31

[19] P.K. Suri and M. Satiza, *System Progress Estimation in Time based Coordinated Checkpointing Protocols*, International Journal of Computer Applications, 52(11): 1–6, 2012.

[20] N. Neves and W.K. Fuchs, *Adaptive Recovery for Mobile Environments*, Communications of the ACM, 40(1): 68–74, 1997.

[21] C–Y. Lin, S–C. Wang and S–Y. Kuo, *An Efficient Time–Based Checkpointing Protocol for Mobile Computing Systems over Mobile IP*, Mobile Networks and Applications, 8(6): 687–697, 2003.

[22] K. Singh, *On mobile checkpointing using index and time together*, World Academy Science, Engineering and Technology, Vol. 26, 2007, pp. 144–151.

[23] 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 (LNEE), Vol. 131, 2013, pp. 173–182.

[24] M. Tripathy and C.R. Tripathy, *A new co–ordinated checkpointing and rollback recovery scheme for distributed shared memory clusters*, International Journal of Distributed and Parallel Systems (IJDPS), 2(1): 49–58, 2011.

[25] B. Gupta and S. Rahimi, *A Novel Low-Overhead Recovery Approach for Distributed Systems*, Journal of Computer Systems, Networks and Communications, Vol. 2009, 2009, pp. 1–8.

[26] J. Fu, M. Min, R. Latham and C. D. Carothers, *Parallel I/O Performance for Application–Level Checkpointing on the Blue Gene/P System*, Workshop on Interfaces and Architectures for Scientific Data Storage (IASDS), in conjunction with IEEE International Conference on Cluster Computing (Cluster), 26–30 Sept. 2011, Austin, TX, pp. 465–473.

[27] I.I. Yusuf and H. W.Schmidt , *Parameterised Architectural Patterns for Providing Cloud Service Fault Tolerance with Accurate Costings*. In Proceedings of the 16th International ACM SIGSOFT symposium on Component–based software engineering (CBSE13), June 17–21, 2013, Vancouver, BC, Canada, pp. 121–130.

[28] L. Ramakrishnan and D. A. Reed, *Performability Modeling for Scheduling and Fault Tolerance Strategies for Scientific Workflows*. In Proceedings of the 17th international symposium on High performance distributed computing (HPC08), 23–27 June 2008, Boston, MA, USA, pp. 23–34.

[29] M. Wang, L. Zhu and J. Chen, *Risk–aware Checkpoint Selection in Cloud-based Scientific Workflow*, Second International Conference on Cloud and Green Computing (CGC2012), Xiangtan, Hunan, China, November 1–3, 2012, pp. 137–144.

[30] E. Caron, F. Desprez, D. Loureiro, and A. Muresan, *Cloud computing resource management through a grid middleware: A case study with DIET and eucalyptus*, IEEE International Conference on Cloud computing (CLOUD2009), 21–25 Sept. 2009, Bangalore, India, pp. 151–154.

[31] M. Sudha and M. Monica, *Investigation on Efficient Management of workflows in cloud computing Environment*, International Journal on Computer Science and Engineering (IJCSE), 2(5): 1841–1845, 2010.

[32] W. Chen and E. Deelman, *WorkflowSim: A Toolkit for Simulating Scientific Workflows in Distributed Environments*. In Proceedings of the IEEE 8th International Conference on E–Science (E–SCIENCE '12), October 8–12, 2012, Chicago, IL, USA, pp. 1–8.

[33] R. N. Calheiros and R Buyya, *Meeting Deadlines of Scientific Workflows in Public Clouds with Tasks Replication*, IEEE Transactions on Parallel and Distributed Systems (TPDS), No. 99, 2013, pp. 1–10.

[34] A. Gupta, L. Kale, F. Gioachin, V. March, C. Suen, B. Lee, P. Faraboschi, R. Kaufmann and D. Milojicic, *The Who, What, Why and How of High Performance Computing Applications in the Cloud*, Technical Reports, HP Laboratories, HPL–2013–49, 2013. pp. 1841–1845.

[35] K.L. Keville, R. Garg, D.J. Yates, K. Arya and G. Cooperman, *Towards Fault–Tolerant Energy Efficient High Performance Computing in the Cloud*, IEEE International Conference on Cluster Computing (Cluster12), 24–28 Sept. 2012, Beijing, pp. 622–626.

[36] B. Nicolae and F. Cappello, *BlobCR: Virtual Disk Based Checkpoint–Restart for HPC Applications on IaaS Clouds*,

Journal of Parallel and Distributed Computing , 73(5): 698–711, 2013.
[37] P. Zaspel and M. Griebel, *Massively parallel fluid simulations on Amazons HPC cloud*, First International Symposium on Network Cloud Computing and Applications (NCCA2011), 21–23 Nov. 2011, Toulouse, France, pp. 73–78.
[38] E. Walker, *Benchmarking Amazon EC2 for high–performance scientific computing*, LOGIN , 33(5): 18–23, 2008.
[39] L. Ramakrishnan, P.T. Zbiegel, S. Campbell, R. Bradshaw, R.S. Canon, S. Coghlan, I.Sakrejda, N. Desai, T. Declerck, and A. Liu, *Magellan: experiences from a science cloud*, In Proceedings of the 2nd international workshop on Scientific cloud computing (ScienceCloud11), June 8, 2011, San Jose, California, USA, pp. 49–58.
[40] B. Meroufel, G. Belalem, *Collaborative Services for Fault Tolerance in Hierarchical Data Grid*, International Journal of Distributed Systems and Technologies (IJDST), 5(1): 1–21, 2014.

Edited by: Florin Fortiş
Received: Mar 1, 2014
Accepted: Jun 24, 2014