Designing a Virtual Machine Checkpointing for IaaS Cloud

Win Win Naing
University of Computer Studies, Yangon (UCSY)
Myanmar
winwinnaing89@gmail.com

Thinn Thu Naing
University of Computer Studies, Yangon (UCSY)
Myanmar
thinnthu@gmail.com

***Abstract*—Nowadays, high availability is one of the critical issues in fault tolerance for cloud data centre. Checkpoint/Restart mechanism is a classical approach for achieving high availability in a cloud computing environment. It is used to provide high availability in various levels such as *application-level, virtual machine-level, and compute node-level*. In this paper, virtual machine (VM)-level checkpointing mechanism for IaaS cloud is proposed. In the VM-level checkpointing scheme, the states of virtual machine are periodically saved on a stable storage or a file. When a VM failure occurs, the system restarts its VM execution from a previously established point on the same machine or another machine. The proposed system is based on the coordinated checkpointing protocol and uses the vector timestamp to get globally consistent state among virtual machines. Moreover, the proposed system can reduce the checkpointing overhead by storing the checkpointing data to the local disk of the host machine.**

***Keywords*—cloud computing, virtual machine, checkpointing, vector timestamp, availability, fault tolerance**

I. INTRODUCTION

Cloud computing become from an evolution of the widespread adoption of virtualization, service-oriented architecture, utility computing and grid computing. Cloud computing platform can offers on demand services with elasticity, scalability on a simple pay-per-use manner. Cloud users or end users not need to know the location and other details of the computing infrastructure that support their computing activities.

Although cloud computing has been widely used in various industry, still there are many research issues to be fully addressed like fault tolerance, workflow scheduling, workflow management, security and privacy etc. Fault tolerance is one of the key issues among all. It is concerned with all the techniques necessary to enable a system to tolerate software or hardware faults remaining in the system after its development. When a fault occurs, these techniques provide mechanisms to the system to prevent system failure occurrence. The main benefits of implementing fault tolerance in cloud computing include failure recovery, lower cost, improved performance metrics etc [1].

A. Motivation

The following motivations encourage doing our work.

First, to simplify the creation of private clouds, many researchers have been proposed several open-source IaaS cloud management frameworks. But all these system do not emphasized on virtual machine level failure. This fact induces to implement our solution.

Second, cloud providers need to guarantee their service level agreements that means their system can solve hardware or software failure for the cloud customers. In this study, we proposed a VM-level checkpointing mechanism that can solve VM failure and guarantees VM-level fault tolerance.

Third, the uses of coordinated checkpointing are very common approach in fault tolerance system to get system wide consistent state. The proposed system uses the vector timestamp-based coordinated checkpointing protocol to guarantee the system globally consistent state.

B. Paper Outline

This paper organizes as follows. Section 2 discusses related works for the proposed system. Section 3 presents the failure model, algorithm and implementation of the proposed system. Section 4 shows the evaluation result of the system. And then Section 5 gives the conclusions. Finally, future directions are discussed in Section 6.

II. RELATED WORKS

A. Fault Tolerance Management in Cloud

Fault tolerance means the ability of a system to respond gracefully to an unexpected hardware or software failure. In [2], A. Tchana, L. Broto and D. Hagimont described that there are three types of failure in cloud computing environment: application failure, virtual machine (VM) failure and hardware failure. Therefore, they explained that there are three possible fault tolerance solutions: application fault tolerance, VM fault tolerance and physical machine fault tolerance. In this paper, the proposed system is emphasized on VM fault tolerance.

B. Levels of Checkpointing

Checkpoint/Restart system may be implemented in three ways: *application-level*, *library-level* and *system-level* [8]. Programmers may implement checkpointing mechanisms at *application-level* where they insert codes to save immediate results of their programs to checkpoint files [9]. Although this approach gives the highest efficiency, it may not be possible

for an automated system to restart an arbitrary application. Programmer can also use *user-level* checkpointing by linking their object codes to a checkpointing library [11] [12].This approach may not require source modifications to the underlying application. But the library imposes restrictions on which system calls the application may use. Alternatively, checkpointing mechanisms can be done at *system-level* or *kernel-level*, where the checkpointing mechanisms are integrated into OS kernels such as [15]. This approach allows application to be checkpointed at any time. As a drawback, system implementations have only been provided on a few current systems due to the difficulty of the implementation of the checkpoint/restart at the system level [8] [17].

C. Checkpointing Mechanisms

Many researcher developed various checkpoint-based fault tolerance system for cloud computing. They also used numerous checkpointing techniques to implement their system. B. Nicolae and F. Cappello [4] [5] proposed an efficient checkpoint-restart system for HPC applications. Their proposed system intended for IaaS clouds and the snapshot time of their system is speed up than other system by using VM disk-image multi-snapshotting and multi-deployment. Some researcher uses proactive fault tolerance system [1] to implement their system. E. Feller, L. Rilling and C. Morin [7] proposed a scalable and autonomic VM management framework for private clouds also called snooze. They used self-organizing hierarchical architecture and apache zookeeper leadership algorithm to develop the fault tolerant virtual machine management framework for cloud computing. Their system provides the fault tolerance at all level of the hierarchy. Moreover I. Goiri and F. Julia [10] developed a smart checkpoint infrastructure for virtualized service providers. They used Another Union File system to differentiate read-only from read-write parts in the VM image.

In Checkpoint-based fault tolerance system, checkpoint overhead and latency is needed to be considered specifically. D. Singh, J. Singh and A. Chhabra [6] evaluated overhead of integrated multilevel checkpointing algorithm in cloud computing. Integrated checkpointing algorithms and load balancing are used to provide high availability to the demand of cloud. Furthermore, N. Limrungsi and J. Zhao [13] applied peer-to-peer checkpointing to develop a novel method for providing reliability as an elastic and on-demand service.

D. Virtual Machine Checkpointing

There are two types of VM checkpointing techniques. They are stop-and-save checkpointing and checkpointing through live migration. The first technique suspends the running virtual machine, copies its entire state and communication information between virtual machines and finally restarts the VM. To recover from a VM failure, the system restarts its execution from a previous error-free, consistent global state recorded by the checkpoints of all virtual machine. In [17], they proposed the Threaded-based Live Checkpointing mechanism (TLC). In their system, they create checkpointing thread that saves the virtual machine state to persistent storage. At the same time, they run virtual machine thread that is allowed to progress with normal execution. As an experiment, their approach can provide high levels of virtual machine responsiveness during checkpointing.

In [3], B. Cully, G. Lefebvre and D. Meyer present another VM-level checkpointing mechanism that is used the checkpointing technique through live migration. Their system provides high availability by propagating frequent checkpoints of an active VM to a backup physical host. Their implementation is based on the Xen virtual machine monitor.

In VM fault tolerance, some researchers tend to reduce virtual machine (VM) downtime. In addition, an existing work that is closely related to ours system is the globally consistent checkpointing [14]. The main purpose of their system is to reduce VM downtime in Virtual Clusters.

III. VIRTUAL MACHINE CHECKPOINTING MECHANISM

Before In this section, we present the failure model, algorithm and implementation of the VM-level checkpointing system for IaaS cloud architecture.

A. Failure Model

The general idea of the proposed system is to handle the crash failure of any virtual machine. The proposed system periodically saves the snapshot of the VM to the local disk of the host machine. When one or more of the VM is crashed, the system resumes VM computation on the same machine by restoring state information from the local disk. But it cann't solves software failures that run over the VMs and damage of the host machine. The system model consists of three main players:

1. **Virtual Machine Monitor (VMM)** is a software program which is a main component of the VM-level checkpointing. It performs as a coordinator in the coordinated checkpointing protocol to checkpoint the snapshot of the VM. Moreover, VM failures can be detected and repaired by VMM.
2. **Virtual Machine (VM)** acts as a checkpointing process in coordinated checkpointing protocol.
3. **Persistent Storage** is a storage server where snapshots of all virtual machines are persistently stored. In this paper, the system uses the local disk of the host machine as a storage.

B. Coordinated Checkpointing using Vector Timestamp

The proposed system uses the vector timestamp-based coordinated checkpointing protocol to implement the checkpointing process. The proposed algorithm is based on the principal coordinated checkpointing protocol using vector timestamp [16].

1) Definitions of Notations

In this protocol, there are n number of executing processes and one process plays the coordinator role and the remaining processes play the checkpointing processes role. Table I shows the definitions of notations that use in the proposed coordination protocol.

2) Vector Timestamp

TABLE I. DEFINITIONS OF NOTATIONS

Notation	Definition
P	Set of processes executing coordinated checkpointing protocol
$p_i \in P (i = 1,2,\ldots,n)$	Checkpointing processes
$p_c \in P$	Coordinator process
$CP(s) \subseteq P (s=1, 2, 3\ldots)$	Set of processes executing the s-th checkpointing
$CPF(s) \subseteq P (s=1, 2, 3\ldots)$	Set of processes which have finished the s-th checkpointing
$send_i(m)$	Event that p_i sends message m
$receive_i(m)$	Event that p_i receives message m
$send_c(m)$	Event that coordinator p_c sends message m
$receive_c(m)$	Event that coordinator p_c receives message m
$checkpoint_i(s)$	Event that p_i executes the s-th checkpointing

After describing the definitions of notations, we also define a few important definitions for the vector timestamp as follows:

Definition 1.

Let VT_i be the vector timestamp of process p_i where length of VT_i is n+1. Given a set of processes $\{p_1 \ldots p_n\}$ belongs to a message vm_i in which VT_i is attached.

Definition 2. (Property of VT_i)

Let VT[a] and VT[b] be the vector timestamps of process a and b where VT[a] < VT[b] if and only if a →b.

Definition 3. (Property of VT_i)

Let $VT_i[i]$ be the occurrence number of events at process p_i.

Definition 4. (Property of VT_i)

For $VT_i[j]$ = k, process p_i recognizes that k events occurred at process p_j.

Definition 5.

Given a set of processes $\{p_1 \ldots p_n\}$ become member of CPF(s) when vector timestamps of $\{p_1 \ldots p_n\}$ are reached to 5.

C. Coordination Protocol

In this paper, we enhanced the previous protocol [16]. In [16], they mentioned that the checkpointing data are recorded at step 4. However, their proposed system didn't consider storing the checkpointing data permanently. Therefore, we extended the algorithm by adding the step 7 in which checkpointing data are stored to the local disk of the host machine.

Fig. 1 shows the detail processes of the proposed coordination protocol. And the remaining notations and detail functions can be found in [16].

Coordinator process p_c holds a message vm_c in which VT_c is attached.

1. The coordinator process p_c executes **sendAction()** and requires checkpointing to each process p_i. $send_c(vm_c) \rightarrow receive_i(vm_c)$
2. Once process p_i receives message vm_c, process p_i executes **receiveAction()**.
3. Process p_i executes **checkpointAction()** and starts $checkpoint_i(s)$. $receive_i(vm_c) \rightarrow checkpoint_i(s)$ where $p_i \in$ CP(s).
4. After process p_i records checkpointing, process p_i executes **sendAction ()** and sends message vm_i to the coordinator. $send_i(vm_c) \rightarrow receive_c(vm_c)$ and $\neg (p_i \in$ CP(s)) $\wedge (p_i \in$ CPF (s)).
5. As soon as the coordinator executes **receiveAction()**, the coordinator sends message vm_c to the process p_i. $send_c(vm_c) \rightarrow receive_i(vm_c)$
6. Once process p_i receives message vm_c, it executes **receiveAction().**
7. And then process p_i performs **saveAction(),** $VT_i[i]$ = 5 is satisfied.
8. Process p_i restarts blocked computing. At this time, $send_i(vm_i) \rightarrow receive_k(vm_i)$ is permitted for process p_k where $VT_i[k] = 4$.
9. Once $VT_c[i]$=2n+1 is satisfied, the s-th checkpointing ends.

Figure 1. The proposed coordination protocol

D. The Checkpointing Phase

In fault tolerance strategy, there are two distinct phases to implement the VM-level checkpointing. They are *checkpointing phase* and *recovery phase*. In the virtual machine checkpointing phase, firstly we need to checkpoint the VMs' state information to handle the case of VM failure. According to the coordinated checkpointing protocol, we use the Virtual Machine Monitor (VMM) as a coordinator process and Virtual Machine (VM) as a checkpointing process. Fig. 2 shows the detail checkpointing process of the proposed mechanism with an example.

In the example of the proposed system, there are one Virtual Machine Monitor (VMM) and four Virtual Machines VM1, VM2, VM3 and VM4 with dependency vectors VT_{VMM} $VT_{VM1}, VT_{VM2}, VT_{VM3}, VT_{VM4}$. Firstly, their all dependency vectors are initialized to (00000) respectively.

Step 1: Virtual Machine Monitor (VMM) performs a start up operation by sending the checkpoint request that includes VMM's vector timestamp to all VMs. VMM increases its vector timestamp (00000) to (00001) before sending the checkpoint request.

Step 2: When a VM1 receives checkpoint request from the VMM, it compares its vector timestamp (00000) with received VMM's vector timestamp (00001) and take the maximum timestamp (00001). And then VM1 increase its vector timestamp to (10001). After that VM1 takes their own checkpoints and increase vector timestamp again. So, vector timestamp of the VM1 becomes (20001). After finishing the checkpointing process, VM1 increases its timestamp to (30001) and send acknowledge message back to the VMM that contains its timestamp. The remaining virtual machines VM2, VM3 and VM4 are also perform the same process as VM1. Therefore, after finishing the step 1 process, vector timestamps of VM2, VM3 and VM4 become (03001), (00301), and (00031) respectively.

Step 3: As soon as the VMM receives acknowledgement messages from VM1, it compares its timestamp (00001) and received timestamp from the VM1 (30001) and takes the maximum timestamp (30001). At the same time, it increases its vector timestamp (30001) to (30002). And then it sends the checkpoint done message to all VMs. VMM increases its vector timestamp (30002) to (30003) before sending the checkpoint done message. When the step 3 finished, the vector timestamp of the VMM becomes (33339).

Step 4: Once VM1 receives checkpoint done message from the VMM, VM1 performs the timestamp comparing process again and takes maximum timestamp and increment it by one to (40003). Finally, VM1 stores its snapshot data to the local disk of the host machine. Finally, it increases vector timestamps (40003) to (50003). According to the coordinated checkpointing protocol, all VMs will be successfully finished their checkpointing process when their amount of timestamps are reached to 5 as VT_{VM1} (50003), VT_{VM2}(35005),VT_{VM3}(33507), VT_{VM4}(33359).

E. The Recovery Phase

Rollback recovery is a process that restores the system back to a consistent state when a failure occurs. In the proposed system, we can face one or more failures at the same time. Fig. 3 illustrates the sequence diagram for how to handle if a failure occurs.

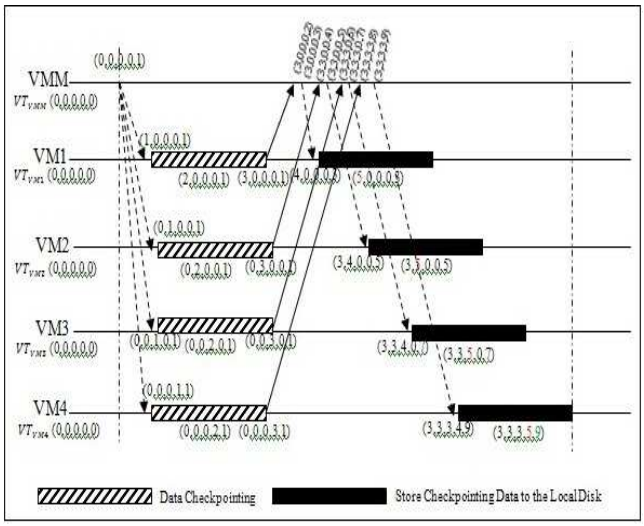

Figure 2. Virtual machine checkpointing

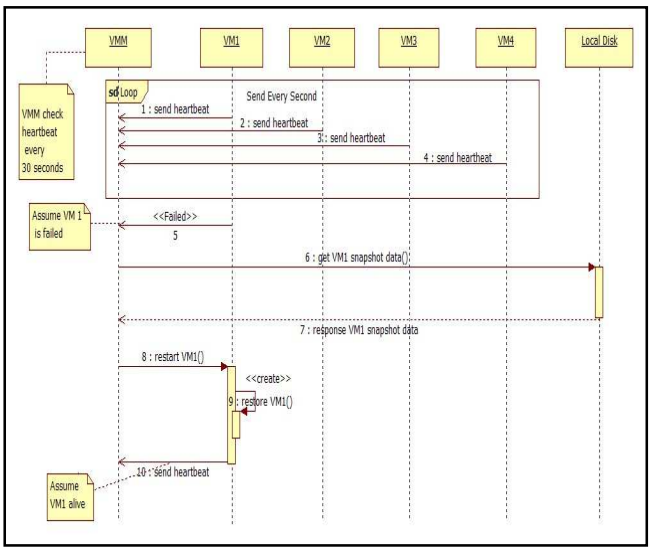

Figure 3. Sequence diagram for virtual machine failure recovery

According to the figure 3, all VMs must be sending their heartbeat messages to the VMM in every second. VMM always checks the heartbeat messages of all VMs at every defined period such as 10 seconds, 20 seconds or 30 seconds etc. At that time, one of the VM's heartbeat messages is lost; we assume that VM is failed. In the figure, we can see that VM1 is failed. And then VMM request the failed VM's snapshot data from the local disk of the host machine. When VMM get the data, it instructs the VM1 to restore on the same machine. After finishing the restoring process, VM1 send heartbeat message back to the VMM. Then, VMM can assume that VM1 is alive when it receives its heartbeat message.

F. Correctness Proof

The proposed system is allowed to guarantee the recovery line generated by the coordinated checkpointing using vector timestamp is globally consistent.

Proof: After finishing the step 6, all virtual machines would be received done message from the virtual machine monitor. And then all VMs would be stored the checkpointing data to the local disk at step 7. At that time, vector timestamps of all VMs would be 5 and they would be satisfied $p_i \in CPF(s)$. Therefore, we can assume all checkpointing data would be consistently stored to the local disk when vector timestamps of all VMs are reached to 5.

IV. EVALUATION AND RESULTS

A. Experimental Environment

Our experiments were conducted on VMware Workstation version 6.0.2. The proposed system used the virtual machine Image that obtains in Eucalyptus IaaS cloud environment to evaluate the system performance. Table II presents the VM Image types available in Eucalyptus.

TABLE II. FIVE TYPES OF VIRTUAL MACHINE IMAGE

No	VM Image Type	CPUs	Memory(MB)	Disk (GB)
1	m1.small	1	128	2
2	c1.medium	1	256	5
3	m1.large	2	512	10
4	m1.xlarge	2	1024	20
5	c1.xlarge	4	2048	20

1) Checkpoint Size

Firstly, we configure one virtual machine for m1.small to have 128 memory, one processor CPU and 2 GB hard disk to know the amount of memory that required saving the snapshot of the VM. We evaluate the remaining types of VM are also tested in this experiment. Fig. 4 shows the VM checkpoint size with different VM Image types that are available in Eucalyptus.

According to the figure 4, m1.small VM image type requires 129 MB for storing one snapshot, c1.medium requires 257 MB, m1.large requires 514 MB, m1.xlarge requires 1024MB and c1.xlarge requires 2048 MB respectively. As we observed that the checkpoint size are directly depended on the VM memory configuration and do not depend on other configuration factors such as CPUs and disk amount.

V. CONCLUSIONS

This paper has proposed a VM-level checkpointing technique for IaaS cloud environment to guarantee the fault tolerance. The main purpose of the proposed system is to provide high availability to the IaaS cloud clients. The technique of the proposed system is based on coordinated checkpointing protocol using vector timestamp. Therefore, the recovery line generated by the coordinated checkpointing using vector timestamp is globally consistent and can reduce checkpointing overhead.

VI. FUTURE DIRECTIONS

This section discusses a number of directions that we intend to explore the proposed system. As we have demonstrated in the previous section, we intended to implement the VM-level checkpointing system for IaaS cloud. In future, we will evaluate the checkpointing overhead, latency, and restart time of the proposed coordinated checkpointing protocol. However, the proposed VM-level checkpointing mechanism cann't solves the software failure that run over the VMs and damage of the host machine. Therefore, we intend to implement the Compute Node (CN)-level fault tolerance system that can solve the damage of the host machine. When the compute node failure occurs, the CN-level checkpointing mechanism will be recovered the failed compute node on the standby node by restoring the checkpointing data of the failed compute node.

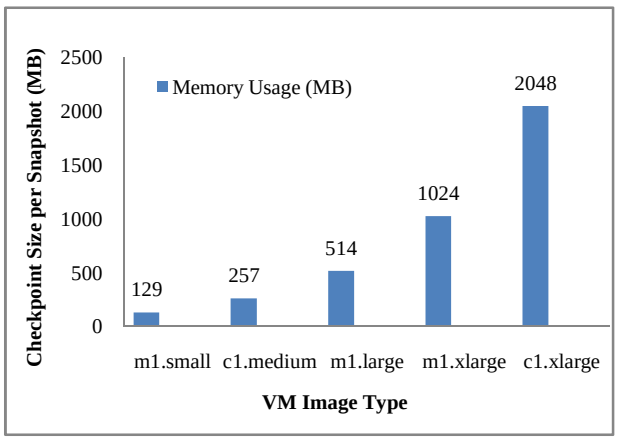

Figure 4. The VM checkpoint size with different VM image type

REFERENCES

[1] A. Bala and I. Chana, "Fault tolerance- challenges, techniques and implementation in cloud computing", Thapar University, 2012.

[2] A. Techana, L. Broto, and D. Hagimont, "Fault tolerant approaches in cloud computing infrastructures", ICAS, 2012.

[3] B. Cully, G. Lefebvre and D. Meyer, "Remus: high availability via asynchronous virtual machine replication", The University of British Columbia, NSDI'08: 5th USENIX Symposium on Networked System Design and Implementation, 2008.

[4] B. Nicolae and F. Cappello, "BlobCR: efficient checkpoint-restart for HPC applications on IaaS clouds using virtual disk image snapshots", *SC'11*,Washington USA, November, 2011.

[5] B. Nicolae, J. Bresnahan and K. Keahey, "Going back and forth: efficient multiployment and multishapshotting on clouds", *HDPC'11*, San Jose, California, USA, June, 2011.

[6] D. Singh, J. Singh and A. Chhabra, "Evaluating overheads of integrated multilevel checkpointing algorithms in cloud computing environment", Dept. of Computer Science & Engineering, Guru Nanak Dev University Amritser, Punjab, India, 2012.

[7] E. Feller, L. Rilling, C. Morin, "Snooze: A scalable and autonomic virtual machine management framework for private clouds", December, 2011.

[8] E. Roman, "A survery of checkpoint/restart implementations", Lawrence Berkeley National Laboratory, Berkeley, CA, July, 2002.

[9] G. Bronevetsky, D. Marques and K. Pingali, "Application-level checkpointing for shared memory programs", Department of Computer Science, Cornell University, 2004.

[10] I. Goiri, F. Julia, J. Guitart and J. Torres, "Checkpoint-based fault-tolerant infrastructure for virtualized service providers", Barcelona Supercomputing Center and Technical University of Catalonia, IEEE, 2010.

[11] J. Duell, "The design and implementation of Berkeley lab's linux checkpoint/restart", 2005.

[12] M. Ricker, J. Ansel, G. Cooperman, "Transparent user-level checkpointing for the native POSIX thread library for linux", College of Computer and Information Science, Northeastern University, Boston, 2006.

[13] N. Limrungsi, J. Zhao, Y. Xiang, T. Lan, H. H. Huang and S. Subramaniam, "Providing reliability as an elastic service in cloud computing", Dept. of Electrical and Computer Engineering, George Washington University, Washington, IEEE, 2012

[14] P. Lu, B. Tavindran and C. Kim, "VPC: scalable, low downtime checkpointing for virtual clusters", IEEE, 2012.

[15] R. Gioiosa, J. C. Sancho, S. Jiang and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers", Washington, USA, SC'05, November, 2005.

[16] T. Jinno, T. Kamiya and M. Nagata, "Coordinated checkpointing using vector timestamp in grid computing", Osaka Kyoiku University, Japan, 2006.

[17] V. Siripoonya and K. Chanchio, "Thread-base live checkpointing of virtual machines", Faculty of Science and Technology, Thammasat University, IEEE, 2011.