Global snapshot of a distributed system running on virtual machines

Carlos E. Gómez*‡, Harold E. Castro* and Carlos A. Varela†
*Universidad de Los Andes - Bogotá, Colombia; †Rensselaer Polytechnic Institute - Troy New York, USA;
‡Universidad del Quindío - Armenia, Colombia
ce.gomez10@uniandes.edu.co, hcastro@uniandes.edu.co, cvarela@cs.rpi.edu

***Abstract*—Recently, a new concept called desktop cloud emerged, which was developed to offer cloud computing services on non-dedicated resources. Similarly to cloud computing, desktop clouds are based on virtualization, and like other computational systems, may experience faults at any time. As a consequence, reliability has become a concern for researchers. Fault-tolerance strategies focused on independent virtual machines include snapshots (checkpoints) to resume the execution from a healthy state of a virtual machine on the same or another host, which is trivial because hypervisors provide this function. However, it is not trivial to obtain a global snapshot of a distributed system formed by applications that communicate among them because the concept of global clock does not exist, so it can not be guaranteed that snapshots of each VM will be taken at the same time. Therefore, some protocol is needed to coordinate the participants to obtain a global snapshot. In this paper, we propose a global snapshot protocol called UnaCloud Snapshot for its application in the context of desktop clouds over TCP/IP networks. That differs from other proposals that use a virtual network to inspect and manipulate the traffic circulating among virtual machines making it difficult to apply them to more realistic environments. We obtain a consistent global snapshot for a general distributed system running on virtual machines that maintains the semantics of the system without modifying applications running on virtual machines or hypervisors. A first prototype was developed and the preliminary results of our evaluation are presented.**

***Keywords*-global snapshot; checkpointing; reliability; fault tolerance; global states.**

I. Introduction

Recently, a new concept called *desktop cloud* (DC) emerged, which was developed to offer cloud computing services on non-dedicated resources. According to [1], DC originates by merging the concepts of cloud computing and desktop grid (DG) while [2], [3] suggest that this paradigm originates by combining the advantages of cloud computing and the benefits of volunteer computing (VC). The DC systems goal is to provide cloud computing services without using dedicated resources as traditional cloud infrastructures that require specialized data centers. DC systems take advantage of the unused processing capacity of computers while users perform their daily activities [3], given that users use very few computational resources when they are in interactive sessions [4], [5], [6] or when computers are fully available. DC is a form of distributed processing which provides an inexpensive or free computing platform whose resources can be aggregated for projects such as scientific projects. The infrastructure is created from shared computational resources of participants when they are totally or partially idle.

DCs, like other computer systems, may experience faults at any time, especially because they run on non-dedicated resources. In this paper, we propose a global snapshot protocol called *UnaCloud Snapshot* for its application in the context of DCs. This work aims at increasing the service level of DCs through reliability and fault tolerance to diversify the range of applications that can utilize these systems, and running unattended applications, thereby facilitating the successful completion of Virtual Machine (VM) deployments, especially those whose execution is scheduled for several days or weeks. New applications could be built to run on such infrastructure. Applications can benefit from the ability to provide reliable service despite the volatility of the physical machines, for example, distributed systems formed by applications that communicate among them.

To validate the work, we use UnaCloud [7], the DC of Universidad de Los Andes (Bogotá, Colombia). UnaCloud is a project in constant development, and it has been the technological support for executing Bag-of-Tasks (BoT) applications in research projects and doctoral theses in several areas [8]. However, UnaCloud currently operates in a best effort way and lack of reliability is one of its main weaknesses.

The paper is organized as follows. Section II talks about the background of the paper. Section III includes the related work. Our approach to carry out the research is described in section IV. The preliminary evaluation, the results and discussion are presented in section V. The conclusion and future work are in the final section.

II. Background

In this section, relevant concepts are introduced to contextualize this research work, namely, desktop cloud, UnaCloud, checkpointing and Global Snapshot Algorithms.

A. Desktop Cloud

DC is a computational paradigm that originates by combining VC or DG systems and cloud computing [2], [3]. The goal of a DC system is to make available shared

resources to users for providing cloud computing services without using dedicated resources. Similar to DG or VC, DC systems use idle computing capacity of the participant computers. DC is based on virtualization technologies, and it can provide processing, storage, networks, applications and services from VMs running operating systems with their respective applications. DC systems are designed to provide cloud computing essential services according to [9]: On-demand self-service, network access, resource pooling, rapid elasticity and measured service.

DC and cloud computing providers offer and manage services for their users differently. Traditional providers rely on specialized data centers, while DC systems use non-dedicated resources, with some degree of heterogeneity. In addition, traditional providers offer users a wider range of tools and have more robust solutions enabling them to meet their Service Level Agreements (SLA). Other services available in cloud providers are monitoring and billing. Conversely, DCs have a smaller variety of services, but for certain needs can perform assigned work successfully with lower costs and in some cases completely free [10]. It is important to note that DC is a concept, it is not a product, so its features depend on the implementation. Such is the case of UnaCloud, which will be described in the following section.

B. UnaCloud

UnaCloud is an opportunistic virtualization-based DC developed at Universidad de Los Andes in Bogotá, Colombia [7]. UnaCloud uses the idle resources of computers that are part of the computer rooms managed by the Department of Systems and Computing Engineering in order to support computing needs of research groups at the same university. UnaCloud is an active project in continuous development and corresponds to its own implementation, unlike most similar systems based-on BOINC, the most popular volunteer computing system.

The UnaCloud testbed consists of three computer labs with 105 computers for undergraduate and master's programs. Computers that are currently serving students have recent technology. The aggregated capacity is significant and it could be used for processing and storage [6].

UnaCloud has been used for the execution of BoT applications, which are independent and have no communication among them. BoT are applications where the problem space is divided into smaller tasks that can be executed in parallel [11]. They are temporary applications whose duration is determined by the amount of data that must be processed and can be used in solving scientific problems. To date, research projects and doctoral theses in different areas such as Chemistry, Bioinformatics, Industrial Engineering and Civil Engineering have used UnaCloud [8].

Despite the amount of work carried out on UnaCloud, so far the reliability and fault tolerance have not been considered among their requirements, since UnaCloud is a *best effort* service. Consequently, it has made little effort to record faults that have been presented in UnaCloud deployments. Moreover, in the current implementation, a monitoring system is in development process. It records resource consumption and keeps track of the hosts that are running VMs. However, it is the user's responsibility to track VMs that are running and restore them manually in case of fault. Therefore, having a fault tolerance mechanism in UnaCloud allows it to be extended to increase the range of applications that can be run, and an automatic fault recovery function will facilitate the successful completion of a distributed system running on VMs.

C. Checkpointing

Checkpointing is a fault tolerance strategy that has two components: a proactive component called checkpoint and a reactive one called restart. The checkpoint component saves the state of a system to resume it from that state when a fail occurs without having to start again from the very beginning. Naturally, if the system is resumed from a checkpoint, nothing performed on the system after the checkpoint was taken is included, so those modifications will be lost. Checkpointing can be full or incremental [12]. The first is the procedure through which every time it is invoked, it saves the complete state of the system. The second saves the system state from modifications after the last checkpoint instead of saving the entire checkpoint. So, it is necessary to keep the files of all previous checkpoints to be possible to resume the execution.

Several techniques to checkpoint distributed applications have been used by different researchers, namely, application-level, library-level, system-level, and virtualization-level checkpointing [13], [14], [15].

Application-level checkpointing is used when the source code of the application and the developers are available. In this case, developers must add lines of code to program the checkpoint and restart functionalities and verify that these modifications do not affect the rest of the application. This technique is not transparent to applications because it requires modify the application code.

Library-level checkpointing functions linking the application code to a checkpointing library such as BLCR[1] [13], [16]. This approach does not force to modify the application, but is strongly coupled with a specific environment [13]. As a result, portability is not the best attribute of the library-level checkpointing. However, its main advantage is that the overhead is marginal.

System-level checkpointing is an approach that rely on modifications made to the kernel of the operating system or the load of certain modules. This solution is very specific and less portable than the previous one, and needs to have

[1]Berkeley Lab Checkpoint/Restart

staff with specialized knowledge and consider particularities of the environment employed [13], [15].

Virtualization-level checkpointing is a functionality provided by all hypervisors, in which the entire state of a VM is stored in a storage medium to form a checkpoint with the possibility of using it later at any time to resume its execution from that state. A VM checkpoint stores all disks attached to the VM, the memory state, and the metadata needed to configure the VM at the moment of resuming its execution [17]. Although this technique is flexible, and it is possible to take a checkpoint from one machine and restart it on another, portability of checkpoints among hypervisors is not available due the lack of standardization.

D. Global Snapshot Algorithms

Global snapshot algorithms are essential for distributed computing. Chandy and Lamport [18] presented the concept of distributed snapshots to determine a global state of a distributed system from the states of the processes (local states) and the states of communication channels among the processes.

Since in a distributed system the concepts of globally shared memory or global clock do not exist, and the network causes unavoidable and unpredictable delays, local states of the processes in distributed system are not recorded at the same time [19]. Therefore, to obtain a global state, some coordination among the participants is required [20].

1) Notation and definitions: A global state (GS) of a distributed system, consisting of n processes and communication channels among them, is a set of the local states (LS) of the processes along with the channel states (CS) [18]. That is, $GS = \{LS_i \cup CS_{i,j}\}$ where $0 \leq i, j < n$ represent the processes in the system, LS_i represents the local state of the process i, and $CS_{i,j}$ represents the state of the unidirectional channel between the processes i and j.

Whenever a message sending or receiving event or internal event occurs, the state of a process changes. Therefore, the state of a process at a given moment is the consequence of the events executed until that moment. If any event occurs later, it does not belong to that state. A message is in transit (it belongs to the channel state) if it has been sent by a process and has not yet been delivered to the destination.

According to [19], there are two conditions required for a global state to be consistent:

- If a message that has been sent by process i to process j belongs to the local state of the sender process, then either the message is part of the local state of the channel that goes from i to j (it is in the channel), or the message is received by the receiver process and is part of its local state.
- If a message that has been sent by process i does not belong to the local state of the sender process (it was sent after recording LS_i), neither the message is part of the channel state from i to j, nor the message received is part of the receiver's local state.

In according to [14], a consistent global state would be the set of the states of the distributed processes and channels if they were recorded at the same time, which is not possible. Instead, authors of [14] uses the concept of "a possible causally consistent global state" that can be obtained upon a consistent cut of events. A cut of events can be seen graphically in a time diagram. In a time diagram, each process has a timeline that is drawn as a horizontal line where time progresses from left to right. Points represent events and arrows represent messages that are sent between two processes. A cut is a sequence of events (one per process) where each event (called cutting events) divides the process history into two parts, the past and the future. Thus, some events occurred before the cutting event and others occur later. A cut is consistent if there are no messages sent in the future that reach their destination in the past [14],[20].

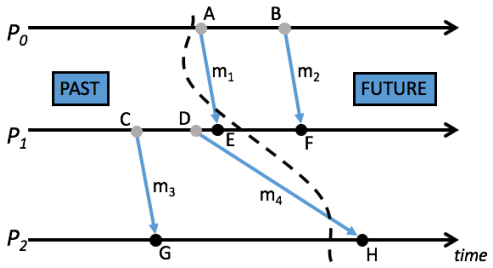

Figure 1. Cut of events

Figure 1 shows an example of a cut of events for a distributed system formed by three processes: P_0, P_1 and P_2. Dashed line represents the cut of events. This picture does not correspond to a consistent cut of events because m_1 is sent from P_0 after the cutting of events and m_1 is received by P_1 before. In this picture, m_1 is an example of a message sent in the future and received in the past.

The challenge here is to determine not only which messages are included in each local snapshot but also when each process should do so, to satisfy the conditions required for a global state to be consistent [19].

To obtain a consistent global snapshot, there are two main approaches: Chandy and Lamport's algorithm for FIFO[2] communication channels [18] and Mattern's algorithm for Non-FIFO communication channels [20], which are described as follows.

2) Chandy and Lamport's Global Snapshot Algorithm: In 1985, Chandy and Lamport proposed their recognized algorithm for computing a consistent global state of a distributed system at runtime with FIFO communication channels [18]. According to the authors, a global state consists of the local states of the processes and the communication channels among them.

[2]Firt-In Firt-Out

The algorithm assumes that:

- There are n processes forming a completely connected network, so each pair of processes have two unidirectional reliable and FIFO channels for communication between them, so all messages arrive correct, only once, and in-order.
- Any process can initiate the execution of the algorithm without stopping the processes.

The general idea of the Chandy and Lamport's algorithm is that a process records its local state and immediately sends a special message called marker to all other processes. A marker establishes the boundary between messages that are sent before and after a process records its local state.

When a process receives a marker:

- If the process has not recorded its local state: The process records its local state; the process records the state of the channel through which it received the marker as an empty message sequence; the process starts recording the messages that arrive through the other channels; and the process sends a marker to all other processes in the system.
- If the process had already recorded its local state, the process records the state of the channel through which it received the marker as the sequence of messages received by that channel until it received the marker.

When all processes have received a marker sent by each of the other processes, the global state is complete. If needed, a storage solution can be used to collect the files to manage them more easily.

3) Mattern Algorithm: On the other hand, Mattern in 1993 proposed an algorithm to calculate a causally consistent global state of a distributed system with non-FIFO channels. The algorithm is based on the Lai and Yang scheme [21] that consists of sending state information within the application messages (piggybacked). According to the Lai and Yang scheme, processes and messages use two colors, white, to indicate that a process has not recorded its local state, and red to indicate that it has already done so. This way, at the beginning, each process is white and becomes red after recording its local state. The color of a message is the color of the process that sends it, that is, a message sent by a white process is white and a message sent by a red process is red. In addition, a local snapshot can be taken by a process at any time, whenever before a red message can be received. The color is represented by a bit and is included in the messages that the applications exchange. Mattern introduced a third color that allows the algorithm to be executed repeatedly by always leaving an available color. The colors are represented by the values 0, 1 and 2 in a circular form.

III. Related Work

Several works have been published around the consistent snapshots for entire systems. The most relevant are VCCP [15], based on [18], and VIOLIN Snapshot [14], based on [20], which have adapted these algorithms to obtain global snapshots of distributed systems running on virtual machines, as shown in Figure 2.

Type of channels / Type of hosts	FIFO	Non-FIFO
Physical Machines	Chandy and Lamport	Mattern
Virtual Machines	VCCP	VIOLIN Snapshot

Figure 2. Related Work

VCCP: A Transparent, Coordinated Checkpointing System for Virtualization-based Cluster Computing is a transparent checkpointing system for distributed applications that run on a cluster of VMs. VCCP is inspired by the Chandy and Lamport global snapshot algorithm [18]. This system is supported by a virtual network that guarantees reliability and FIFO communication channels. The virtual network allows VCCP to control data link layer frames that move the messages between adjacent devices. Thereby, the coordination among participant nodes is implemented in the data link layer rather than the application layer. The algorithm has an initiator and includes a pause in the VMs to prevent the transmission of messages while capturing the snapshot.

The contributions of VCCP are: A layered architecture to separate the main functionalities, the checkpointing mechanism to obtain the global snapshot, and the correctness analysis of the solution.

On the other hand, *VIOLIN Snapshot* is a work designed to achieve a consistent global snapshot for a distributed system running on VMs without modifying the applications running on them.

VIOLIN is a virtual network environment that provides the same services of a physical network. VMs are connected via VIOLIN switches, and there is one switch on each host. VIOLIN switches intercept traffic between VMs and send Ethernet frames through UDP[3] tunneling to the destination VM if it is running on another PM.

VIOLIN Snapshot simplifies Mattern's global snapshot algorithm for systems with non-FIFO communication channels [2]. VIOLIN snapshot, similar to VCCP, is supported by a virtual network, which is used to control the data link layer frames. The virtual network is called VIOLIN and formed by VIOLIN switches. The data link frames are intercepted to identify the piggybacked colors in application messages to know if a message was sent before or after that the the local snapshot was taken by the sender VIOLIN switch. After that, the VIOLIN switch forwards the traffic to the destination by UDP tunneling or discards it to prevent a frame that does not belong to the global snapshot reaches its destination if the local snapshot has not been taken.

The contributions of VIOLIN Snapshot are: the adaptation of a classical theoretical algorithm to a practical scenario,

[3]User Datagram Protocol

and the explanation about its correctness.

VCCP shares some characteristics with VIOLIN Snapshot. Both of them are designed to work without modifying the applications running on the VMs. Similarly, VCCP and VIOLIN Snapshot use a custom virtual network through which traffic is intercepted. On the other hand, there are aspects in which they differ. VCCP is inspired by the Chandy and Lamport's algorithm and is oriented to systems with FIFO channels, while VIOLIN snapshot is inspired by the Mattern's algorithm and as a result, it obtains a global snapshot on systems with channels that can not guarantee FIFO property. In addition, VCCP uses control messages called markers, while VIOLIN snapshot uses piggybacking to implement the concept of colors in messages. VCCP pauses execution of the VMs and VIOLIN snapshot does not; and VCCP logic is implemented in nodes while the VIOLIN Snapshot logic is implemented in VIOLIN switches.

We identified some opportunities upon analyzing VIOLIN Snapshot. VIOLIN switches causes additional consumption of resources, a bottleneck that increases latency, and of course, a single point of failure on each host. In addition, when using UDP tunnels to send frames passing from one PM to another, reliability can be affected if the application running on the VMs uses TCP[4] as the transport protocol. The overhead and latency caused by encapsulation is increased. Therefore, considering the valuable work done by [14] in the VIOLIN Snapshot solution, along with the improvement opportunities identified, we are proposing our system of global snapshot, which is presented in next section.

IV. Approach

One of the challenges of DC systems is their ability to provide reliable services running on unreliable infrastructure because their computational resources are non-dedicated, heterogeneous and may present high volatility. In this work, we propose a checkpointing based fault tolerance approach for DC systems that execute distributed systems formed by applications communicating among them.

This work emerges from the need to ensure that the execution of a distributed system on a DC can finalize successfully. This is important because after an execution is interrupted, the work on the compromised VMs can be lost if the DC is not prepared to face this situation. We aim to increase the service level and diversify the range of applications that can utilize these systems. Moreover, it enables running unattended applications on a DC. We obtain a consistent global snapshot for a general distributed system running on VMs that maintains the semantics of the system based on the algorithm used by VIOLIN Snapshot [14]. A first prototype was developed and it is described below.

[4]Transmission Control Protocol

A. Global Snapshot Protocol

Since almost any distributed application can run on VMs, our global snapshot protocol offers a solution with a high degree of generality. However, as these are snapshots of complete VMs, it is unavoidable to assume that it is an expensive solution that will not necessarily be convenient in all cases or for all types of users. The general principle is that neither the applications running on the VMs nor the hypervisor are going to be modified. We use the concept of colors in the node to determine whether or not a network message is part of the global state of the application. So, in our implementation, it is only required to add a small amount of rules to the firewall on each VM. The global snapshot protocol is an external application that runs on the host at the same level as the hypervisor, and we can not assume that the channels are FIFO. As a consequence, we adapted the simplified Mattern algorithm published in [14] for operation on a TCP/IP network. In our case, a node called Coordinator is in charge of controlling the global snapshot algorithm. Each participant must color outgoing messages accordingly if they have been sent before or after obtaining the local snapshot. The idea is to mark those packets using colors in the sender side and recognize these colors in the destination side for discarding, supported by a firewall that runs on the VMs.

Assign a color consists of setting an arbitrary number and introduce it on each outgoing packet, without modifying the message. In this version of our solution, we use the DiffServ field (previously Type of Service) of the IP datagram to carry the specified value.

To modify the packets, we use the *iptables* firewall that is available in the almost every distributions of the Linux operating system. Iptables was configured to set an output rule on the *mangle* table (which is used to modify network packets) to assign the chosen number (the color), for example $0x10$. Each VM is configured to verify that the color (the number assigned) corresponds to the expected value if the local snapshot has been taken or not. This configuration is also done by creating a rule for iptables in charge of examining all incoming packets that have a specified value in the DiffServ field. Each package that meets this rule will be discarded and will not reach its destination. An example of the configuration rules used can be seen in the Figure 3.

```
iptables -t mangle -A OUTPUT -j DSCP --set-dscp 0x10

iptables -A INPUT -m dscp --dscp 0x10 -j DROP
```

Figure 3. Iptables configuration rules

The implementation is a pure peer-to-peer system that depends of a metadata server to keep the network information for exchanging messages among the processes. Any process can be the coordinator, but this role is assumed by the first

process that contacts the metadata server.

Figure 4 shows the global snapshot protocol we developed in this work, in which for simplicity, the metadata server is not shown.

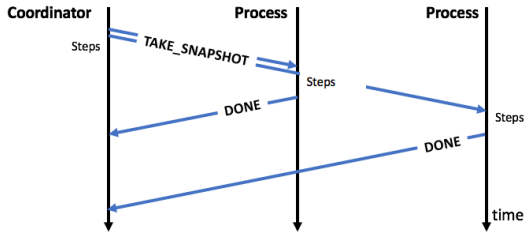

Figure 4. Global Snapshot Protocol

The protocol starts when the coordinator sends a *TAKE_SNAPSHOT* message to the other participants and then it takes the local snapshot. The other processes, upon receiving the message, take the local snapshot, in the same way as the coordinator. Then, they send the *DONE* message to the coordinator, who terminates the protocol upon receiving a *DONE* message from each of the other processes.

Steps in the Figure 4 consist in coloring outgoing packets with the presnapshot color, configure the iptables firewall to prevent delivering packets with postsnapshot color, take the local snapshot, coloring outgoing packets with the postsnapshot color and accept the arrival of postsnapshot packets.

B. Restore Protocol

The restore protocol takes place when a system fault is detected. Its goal is to resume the execution of the distributed system from the last consistent global snapshot stored. It behaves similarly to the global snapshot protocol. The coordinator process sends a *RESTORE* message to the other processes. Then, all processes (including the coordinator) restore the VMs of the system from the last snapshot. Processes send a *RESTORE_DONE* message to the coordinator after restoring the VMs. In the same way as the global snapshot protocol, the restore protocol ends when the coordinator receives a *RESTORE_DONE* message from all other processes.

V. Evaluation, Results and Discussion

This section reports the results of our preliminary evaluation of UnaCloud Snapshot, considering the following metrics: global snapshot time, local snapshot time and overhead. We describe the platform where the tests were carried out. Then, we show the results and the analysis.

We have tested in two different environments. The first one was made using two heterogeneous computers, while for the second one was used a computer room formed by 16 desktops with homogeneous configuration.

- *First environment:* To perform the experiments, we used two computers. A physical server with an Intel(R) Xeon(R) CPU E31220 @ 3.10GHz processor, with 8 GB of RAM and Ubuntu Linux as operating system, and a MacBookPro laptop with a 2,3 GHz Intel Core i5 processor, 16 GB 1600 MHz DDR3 RAM, and MAC OS X. The computers are connected through the RPI field network, and they are located in different buildings. Using Oracle VirtualBox Hypervisor, four identical VMs were installed on each host with 1 CPU core, 650 MB, 20 GB of hard disk and Debian GNU/Linux 8.7 (jessie) as operating system with text mode only.
- *Second environment:* The computer room is composed of 16 desktop computers with Intel (R) Core i5-2300 @ 2.80GHz processor, with 8 GB of RAM and Microsoft Windows 7 as operating system. This computer room is connected by a 1 Gbps local area network. In this case, we used the same hypervisor and VMs as in the first environment.

We developed the distributed system for the experiments. This is a ring token passing, in which one process is running on each VM forming an ordered ring. This is a network and CPU intensive application given that a numbered token is exchanged among processes and it performs a determined amount of float-point operations after a process receives the token. This application has a deterministic outcome, so it allows us to check the correct termination, independently if during the process the global snapshot protocol were performed or not, or if the execution finished upon restoring the system from a global snapshot.

Since the results vary from one execution to another, we realized five executions for each scenario, for each test.

A. Heterogeneous Environment

1) Global Snapshot Time vs Local Snapshot Time: For this test, we executed the distributed system considering three scenarios. 1) Two VMs, one on each host; 2) Four VMs, two on each host; and 3) eight VMs, four on each host.

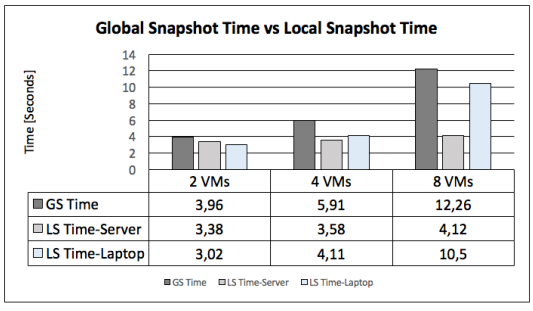

Figure 5. Environment 1: Global Snapshot Time vs Local Snapshot Time

Figure 5 shows the global and local snapshot times experienced by the server and the laptop. We note that the global snapshot time is limited by the slowest computer, in

this case the laptop. In addition, performance degradation is noticeable when the number of VMs increases especially when the laptop performs the local snapshot for 4 VMs. It is important to take into account that in a heterogeneous environment, the coordinator of the global snapshot protocol must be executed on the host with better features because it is necessary to guarantee that its local snapshot is finished before it receives the DONE messages from other hosts.

2) Overhead: To measure the overhead, we executed the ring token passing system considering the same three scenarios with 100,000 and 200,000 tokens exchanged. We were interested in determining the impact of the global snapshot protocol on the distributed system run time. So, we measured five times the total run time with and without the global snapshot, and we analyzed the average.

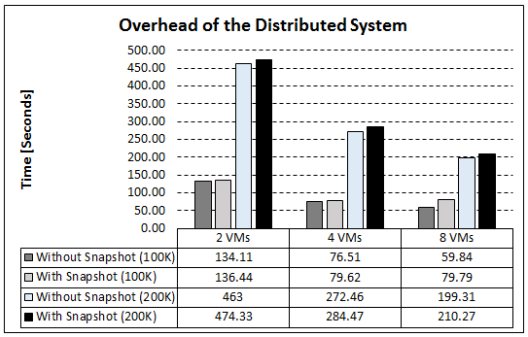

Figure 6. Overhead of the Distributed System

From the results shown in the Figure 6, we note that the overhead is minimal in all scenarios. Also, the overhead is not affected by the duration of the execution of the distributed system. This can be explained since the distributed system consists of intensive applications in network and CPU. Note that when we are running fewer amount VMs on each host, there are more messages that communicate over the network. Therefore, latency impacts the the execution time of the application.

B. Homogeneous Environment

We consider several scenarios using up to four VMs per host and execution of 4, 8 and 16 VMs. We are interested in determining the relationship between the global and the local snapshot times when we are running different combinations of #VMs and the #VMs and #PMs ratio.

1) Global Snapshot Time vs Local Snapshot Time – One VM per host: To compare the Global Snapshot Time and the Local Snapshot Time, we first run the distributed system on 4, 8, and 16 VMs, each running on a host.

Figure 7 shows the global and local snapshot times experienced by the desktops when one VM was used per host for 4, 8, and 16 VMs. The results show a similar behavior in all three cases. In addition, the increase in the number of

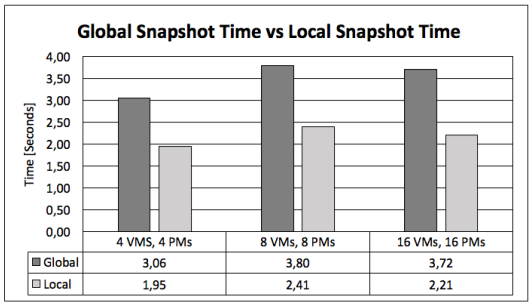

Figure 7. Environment 2: Global Snapshot Time vs Local Snapshot Time

VMs does not represent a considerable increase in the local snapshot time and the global snapshot time.

2) Global Snapshot Time – More than one VM per host: In order to analyze the impact of running our solution in a homogeneous environment when multiple VMs are running at the same time on a host, we tested several combinations for running 4, 8 and 16 VMs. Particularly, we performed an analysis of variance (ANOVA) of two factors with interaction. The two factors are the #VMs and the #VMs running on the same host simultaneously. Table I shows the descriptive information of the tests, including the confidence interval for the global snapshot time.

#VMs and #PMs ratio	Average	Standard Deviation	Lower limit	Upper limit
4	3,65	0,33	3,58	3,71
8	5,15	1,22	4,98	5,31
16	20,92	2,55	20,65	21,19

Table I
SUMMARY OF DESCRIPTIVE STATISTICS

Figure 8 shows the relationship between the global snapshot time and the #VMs and #PMs ratio.

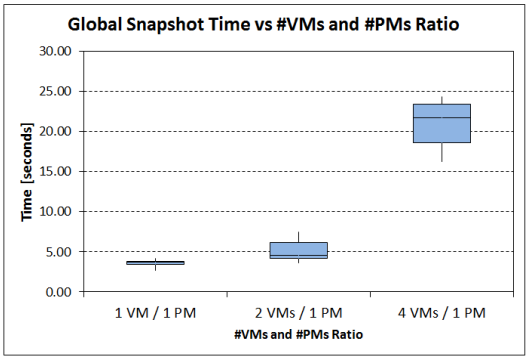

Figure 8. Envirenment 2: Global Snapshot Time by Ratio

The results in Table I and Figure 8 show a high consistency of the data, since a low variation of them with respect to the average value can be appreciated. In addition, we

observe a significant variation in the global snapshot time when the global snapshot is performed while 4 VMs are running on the same host at the same time. This time is mainly due to the competition for the use of the disk when storing the snapshots of each VM. However, there is no important difference if we go from having 1 VM to 2 VMs running on the same PM. This is an interesting result that can be used along with the resource allocation algorithms of UnaCloud. In this way, UnaCloud can prevent to locate more than 2 VMs on the same host, if it is necessary to offer a reliable service to the user.

VI. Conclusion and Future Work

We have presented the first version of our checkpointing based fault tolerance approach to take a global snapshot of a distributed systems formed by applications communicating among them running on a set of virtual machines. We performed experiments for heterogeneous and homogeneous platforms and obtained our first results, which are interesting, particularly for long-term executions on the desktop cloud systems. The experimental results of our preliminary evaluation showed a low overhead and a high data consistency. We found that taking a global snapshot when 4 VMs are running on a host at the same time presents an evident increase in the global snapshot time. This can be used in the resource allocation process of UnaCloud to offer a reliable service for our users.

Since this work is under development, there are many job opportunities to do in the near future. Among them we can highlight the performance analysis of the system restore process from a global snapshot and the development of tests using other platforms to analyze a greater number of VMs and other distributed systems running on VMs.

References

[1] G Fedak. *Contributions to Desktop Grid Computing*. PhD thesis, Ecole Normale Supérieure de Lyon, 2015.

[2] V Cunsolo, Salvatore Distefano, Antonio Puliafito, and Marco Scarpa. Volunteer computing and desktop cloud: The cloud@home paradigm. In *Network Computing and Applications, 2009. NCA 2009. Eighth IEEE International Symposium on*, pages 134–139. IEEE, 2009.

[3] A Alwabel, Robert John Walters, and Gary Brian Wills. A view at desktop clouds. In: ESaaSA 2014.

[4] M Mutka and M Livny. *Profiling workstation's available capacity for remote execution*. University of Wisconsin-Madison. Computer Sciences Department, 1987.

[5] K Ryu. Exploiting idle cycles in networks of workstations. Ph.D. dissertation, University of Maryland, College Park, 2001.

[6] C Gómez, E Díaz, C Forero, E Rosales and H Castro. *Determining the Real Capacity of a Desktop Cloud.* Latin American High Performance Computing Conference. Petropolis, Brazil. Pages 62–72, 2015. Springer, Cham.

[7] E Rosales, H Castro, and M Villamizar. *UnaCloud: Opportunistic cloud computing infrastructure as a service.* Cloud Computing 2011: The Second International Conference on Cloud Computing, GRIDs, and Virtualization. Rome, Italy. Pages 187–194, 2011.

[8] H Castro, E Rosales, and M Villamizar. *Clusters computacionales para la investigación - personalizables, eficientes, amigables y a costo cero*. Revista Ingeniería, (37).

[9] P Mell, et al. The NIST definition of cloud computing. In: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology Gaithersburg, 2011.

[10] A Abdulelah, R Walters and W Gary. Towards a volunteer cloud architecture. In: European Workshop on Performance Engineering, Springer. 248–251, 2012.

[11] D Bakken and R Schlichting. Tolerating failures in the bag-of-tasks programming paradigm. Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium. IEEE. 248–255, 1991.

[12] H Agarwal and A Sharma. A comprehensive survey of fault tolerance techniques in cloud computing. In *2015 International Conference on Computing and Network Communications (CoCoNet)*, pages 408–413. IEEE, 2015.

[13] 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.

[14] A Kangarlou, D Xu, P Ruth, and P Eugster. Taking snapshots of virtual networked environments. In *Proceedings of the 2nd international workshop on Virtualization technology in distributed computing*, page 4. ACM, 2007.

[15] H Ong, N Saragol, K Chanchio, and C Leangsuksun. VCCP: A transparent, coordinated checkpointing system for virtualization-based cluster computing. In *2009 IEEE International Conference on Cluster Computing and Workshops*, IEEE, 2009.

[16] P Hargrove and J Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. *Journal of Physics Conference Series*, vol 46: 494–499, 2006.

[17] Oracle Corporation. *Oracle VM VirtualBox: User Manual Version 5.0.20*, 2016.

[18] K Chandy and L Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[19] A Kshemkalyani and M Singhal. Distributed computing: principles, algorithms, and systems. *Cambridge University Press*, 2011.

[20] F Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[21] T Lai and T Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.