Diskless Checkpointing with Rollback-Dependency Trackability

Raphael Marcos Menderico and Islene Calciolari Garcia
Institute of Computing
State University of Campinas (UNICAMP)
Campinas, SP, Brazil
Email: {rmm, islene}@ic.unicamp.br

***Abstract*—One way to implement fault tolerant applications is storing its current state in stable memory and, when a failure occurs, restart the application from the last global consistent state. If the number of simultaneous failures is expected to be small a diskless checkpointing approach can be used, where a failed process's state can be determined only accessing non-faulty process's memory.**

In the literature diskless checkpointing is usually based on synchronous protocols or properties of the application. In this paper we present a quasi-synchronous diskless checkpointing algorithm, called RDT-Diskless, based on Rollback-Dependency Trackability. The proposed algorithm includes a garbage collection approach that limits the number of checkpoints that must be kept in memory. A framework, called Cheops, was developed and experimental results were obtained from a commercial cloud environment.

***Keywords*-checkpointing; fault-tolerance; availability; dependability; distributed algorithms**

I. Introduction

In recent years people started using the power of computers clusters with thousands of processors [1], either on custom-made interconnected boards [2] or on off-the-shelf computers connected by a high speed network [3]. With a large number of components, it is expected that the mean time between failures will be some hours even if these components are created to be extremely reliable and, independently, have a small failure rate. Although these clusters are usually designed to keep working with some failed components, applications need to be reworked to deal with this situation [4].

Fault tolerant applications can store its current state in stable memory and, when a failure occurs, restart the application from the last consistent global state [5]. These states can be stored on disk or on central stable storage system, which allows any process to recover a failed process' state. However, this approach can increase the execution time, particularly for those who take checkpoints frequently [6]. It is also possible to use a memory-based storage system, where a failed process' state can be determined accessing non-faulty process' memory. This technique, called diskless checkpointing [7], does not support whole-system failures, but it can be used if the number of simultaneous failures is small.

In this paper we present a quasi-synchronous approach for diskless checkpointing. A quasi-synchronous checkpoint algorithm allows the processes to take checkpoints asynchronously, but some additional checkpoints may be induced to guarantee the construction of recovery lines. The proposed algorithm, called RDT-Diskless, is based on Rollback-Dependency Trackability (RDT) [8]. This property allows the use of a garbage collection approach that limits the amount of checkpoints that must be kept in memory [9].

A framework, called Cheops, was developed and simulation results obtained using the Amazon Elastic Compute Cloud (Amazon EC2) and the Amazon Simple Storage Service (Amazon S3)[1]. In these experiments, we have replicated processes's full state. However, our approach could be used with error recovery protocols, such as parity [10], Reed-solomon [11], and *Information dispersal algorithms* [12] in order to reduce the amount of data to be transmitted.

The rest of the paper is divided as follow: Section II presents related work, Section III describes the computational model, Section IV discusses the number of distint repositories and tolerated faults when using diskless checkpointing, Section V explains the RDT-Diskless algorithm, Section VI presents *Cheops* and experimental results. Section VII concludes this article.

II. Related work

There are several studies about diskless checkpointing, most of them either rely on some characteristic of the underlying application [13], [14] or, even though they are designed to be used by any application, they are based on synchronous checkpointing protocols [4], [10], [15]–[19], therefore being more fitted for applications which naturally have a synchronization barrier.

[1] http://aws.amazon.com

1060-9857/10 $26.00 © 2010 IEEE
DOI 10.1109/SRDS.2010.17

IEEE computer society

Plank and Li [10] described the first diskless checkpointing protocol, which takes advantage of memory paging to keep track of the differences among the current and previous checkpoints of a process. A parity mechanism implemented between the process and a special node called checkpoint server assure you are able to tolerate a failure of one single process. A backup server, updated only during a synchronization phase, can be used in case the main mechanism fails.

After that, some studies aim to compare different replication techniques for diskless checkpointing, looking at latency for recovery, overhead during application execution, storage strategies in memory and disks [7], error-correction algorithms and repository strategy [16], [18], [19]. Generally, on synchronous-based diskless checkpointing protocols, local error correction is better than global error correction, the overhead is lower but near disk-based checkpointing and there are huge improvements in the recovery latency time.

III. Computational model and definitions

A distributed system is composed by n processes $(p_1, p_2, \ldots, p_n)$ that can only exchange information through messages. The communication channels are reliable and all messages are delivered in an arbitrary but finite time frame, not necessarily in order. Nodes are subject to crash failures (can fail leaving no data), and processes are subject to crash-recovery failures (can be recovered after a failure). All processes' variables must be stored, and it could be necessary to implement additional recovery systems for non-deterministic data, like user inputs.

Typically, a process is divided in two layers [20]. The application layer is responsible for the distributed computation and sends checkpoint requests to the checkpoint layer. The checkpoint layer collects process' variables and saves checkpoints according to some protocol. Every process has a stable storage associated with it that is not necessary a disk and can be placed in another node. Its implementation depends on the fault tolerance requirements imposed by the designer [20].

A process p_a's execution is treated as an ordered set of events $e_a^0, e_a^1, \ldots$. Every event has an state associated with it composed by all process' variables immediately after the event occurred. A global state of a distributed system is a set composed by one state of each process.

A checkpoint is a state chosen by the process as a candidate to compose a consistent global state [5]. An arbitrary checkpoint α of a process p_a is denoted by c_a^α. A stable checkpoint s_a^α is c_a^α when stored in a way it can be loaded even in case of failures, according to the fault tolerance parameters set on the system. Every process has an initial checkpoint, called c_a^0 and the last stable checkpoint of a process is denoted by s_a^{last}.

We say that $c_a^\alpha \rightarrow c_b^\beta$ if the event that created c_a^α happened before the event that created c_b^β, according to the causal precedence defined by Lamport [21]. If one checkpoint causally precedes another, they cannot be part of the same consistent global checkpoint. However, two checkpoints can be concurrent, but cannot be part of the consistent global checkpoint. The precedence among checkpoints can be captured by the z-precedence relation [22], [23].

Definition 1. Z-precedence between checkpoints: c_a^α z-precedes c_b^β *(*$c_a^\alpha \rightsquigarrow c_b^\beta$*) iff*

- $c_a^\alpha \rightarrow c_b^\beta$, *or*
- $\exists c_c^\gamma : (c_a^\alpha \rightsquigarrow c_c^\gamma) \wedge (c_c^{\gamma-1} \rightsquigarrow c_b^\beta)$.

In the simplest precedence that is not a causal precedence, two messages form a path that resembles the letter Z in a space-time diagram (Figure 1 (a)).

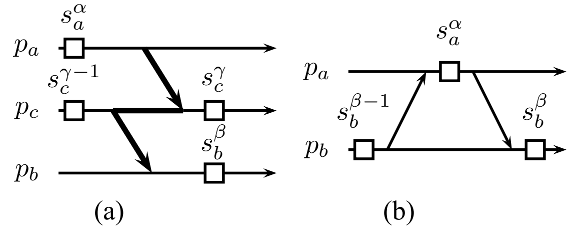

Figure 1. Z-precedence.

A checkpoint that z-precedes itself (Figure 1 (b)) is *useless* and cannot be part of any consistent global checkpoint [23] or recovery line.

Definition 2. *The recovery line of a set F of fault processes is equal to the consistent global checkpoint which discards the lowest number of checkpoints.*

A recovery line, defined as follows, is unique for a single process or for a given set F of fault processes [24].

Definition 3. *Let* s_f^{last} *be the last stable checkpoint of* p_f*. The recovery line of* p_f *is:*

$$R(p_f) = \bigcup_{i=1}^{n} \{c_i^{max(k)} | s_f^{last} \not\rightsquigarrow c_i^k\}$$

Definition 4. *The recovery line of a set F of failed processes is:*

$$R(F) = \bigcap_{p_f \in F} \{R(p_f)\}$$

In this paper we are also going to consider recovery lines based on any stable checkpoint of a process:

Definition 5. *The recovery line of a checkpoint* c_a^α *is:*

$$R(c_a^\alpha) = \bigcup_{i=1}^{n} \{c_i^{max(k)} | c_a^\alpha \not\rightsquigarrow c_i^k\}$$

IV. Distinct repositories and maximum number of failures

In order to isolate the checkpointing management from the actual recovery algorithm we will introduce a classification for information dispersal algorithms according to the number of binary packages produced (ρ) and how many can be discarded without losing the ability to recover the original information (ϕ).

Definition 6. *A (ρ,ϕ)-algorithm must define the following operations:*

- *disperse(c: checkpoint): returns ρ data packages;*
- *rebuild(v: packages[$\rho - \phi$]): returns checkpoint c.*

An algorithm based on replication can be expressed as **(ρ,ρ-1)-replication**:

- *disperse*(c: checkpoint) returns ρ copies of c;
- *rebuild*(v: packages[1]): returns $v[0]$.

An algorithm based on parity using bitwise XOR can be expressed as **(ρ,1)-parity**:

- *disperse*(c: checkpoint) split c into $\rho - 1$ packages, labeled ρ_0, ρ_1, ..., $\rho_{\rho-1}$. Returns $\{\rho_0, \rho_1, \ldots, \rho_{\rho-1}, \rho_0$ **XOR** ρ_1 **XOR** ... **XOR** $\rho_{\rho-1}\}$.
- *rebuild*(v: packages[ρ-1]): returns the concatenation of all original packages in the same order they were split (using parity to recovery one package if necessary).

A. ρ-stable checkpoint

Diskless checkpointing is usually based on synchronous protocols and on the global state of the application [25]. Quasi-synchronous checkpointing protocols work with single checkpoints and determine the global state at recovery time. To extend diskless checkpointing to quasi-synchronous protocols, we will define a ρ-distributed checkpoint:

Definition 7. *Checkpoint c_a^α is ρ-distributed if all the packages produced by the function* ***disperse***(c_a^α) *are stored on distinct repositories.*

A ρ-distributed checkpoint preceded by a non ρ-distributed one (Figure 2).

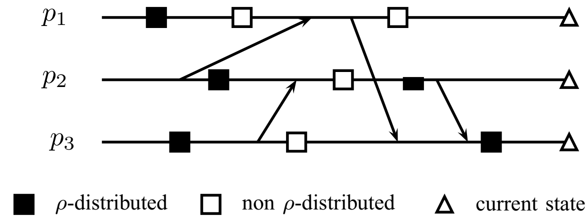

Figure 2. ρ-distributed and non ρ-distributed checkpoints

Definition 8. *A ρ-stable history of a process p_a ($h_\rho(p_a)$) in a consistent cut $\mathscr{C}$ is the set of checkpoints of p_a such that $c_a^\alpha \in h_\rho(p_a)$ if c_a^α is ρ-distributed in any cut $\mathscr{C}' \subseteq \mathscr{C}$ and $\forall \beta < \alpha, c_a^\beta \in h_\rho(a)$.*

Definition 9. *Every $c_a^\alpha \in \mathscr{C}_\rho = h_\rho(p_1) \cup h_\rho(p_2) \cup \cdots \cup h_\rho(p_n)$ is called a ρ-stable checkpoint.*

B. ρ-stable recovery lines

Figure 3 presents a scenario with three processes in which only one of them has not failed. The recovery line of the system is $R(p_2) \bigcap R(p_3)$ that is equivalent to $R(p_2)$. However, even if $\phi = 2$, it is not possible to recover from this failure since the checkpoint for p_3 on $R(p_2)$ is not ρ-distributed. This way, we must focus on ρ-stable checkpoints, that are not only ρ-distributed but also only preceded by other ρ-distributed checkpoints from the same process.

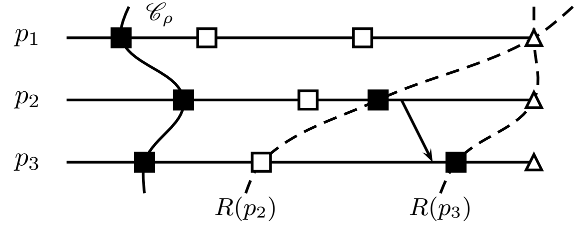

Figure 3. Non-recoverable failure scenario

We must define a new recovery line that includes ρ-stable checkpoints from failed processes. Let $s_{\rho f}^{last}$ be the last ρ-stable checkpoint of a process p_f.

Definition 10. *The ρ-stable recovery line of p_f is*

$$R_\rho(p_f) = R(s_{\rho f}^{last}) = \bigcup_{i=1}^{n}\{c_i^{max(k)} | s_{\rho f}^{last} \not\rightarrow c_i^k\}$$

Definition 11. *The ρ-stable recovery line of a set F of processes in a consistent cut $\mathscr{C}$ is*

$$R_\rho(F) = \bigcap_{p_f \in F}\{R_\rho(p_f)\}$$

The following theorem will show that this recovery line is suitable for diskless checkpointing protocols.

Theorem 1. *Given a set F of failed processes, such that $|F| \leq \phi$, it is always possible to rollback the distributed application to $R_\rho(F)$.*

Proof: Let's assume that checkpoints from non-faulty processes are always available, either because they are already ρ-stable or because its own process is keeping a copy of it while waiting for its stability. From Definition 11, $R_\rho(F)$ is an intersection of recovery lines and must contain only ρ-stable checkpoints from faulty processes. Since $|F| \leq \phi$, it is possible to rebuild these checkpoints and combine them with checkpoints from non-faulty processes in order to rollback to $R_\rho(F)$. ■

C. ρ-needless checkpoints

A *needless* checkpoint does not belong to any current or future recovery line and should be discarded [9], [24]. We need to define ρ-needless checkpoints:

Definition 12. *Checkpoint ${s_\rho}_a^\alpha$ is ρ-needless in a consistent cut $\mathscr{C}$ if it does not belong to any recovery line $R_\rho(p_j)$ for $1 \leq j \leq n$ in $\mathscr{C}$.*

We must prove that ρ-needless checkpoints do not belong to any recovery line in the future of $\mathscr{C}$ and these checkpoints remains ρ-needless in case of a rollback.

Lemma 1. *Let s_a^α be a ρ-needless checkpoint in a cut $\langle s_1^{i_1}, s_2^{i_2}, \ldots, s_n^{i_n}\rangle$. It should remain ρ-needless in any future cut $\langle s_1^{i_1+\epsilon_1}, s_2^{i_2+\epsilon_2}, \ldots, s_n^{i_n+\epsilon_n}\rangle$, $\forall j : \epsilon_j \geq 0$ and $\exists j : \epsilon_j > 0$ for $1 \leq j \leq n$.*

Proof: We know that $s_a^\alpha \notin R(s_j^{i_j})$ and must prove that $s_a^\alpha \notin R(s_j^{i_j+\epsilon_j})$. Let's consider the following cases, assuming that s_a^α is not useless (otherwise it would be trivially ρ-needless).

1) $\forall j, s_j^{i_j} \rightsquigarrow s_a^\alpha$ — Since $s_a^\alpha \in \langle s_1^{i_1}, s_2^{i_2}, \ldots, s_n^{i_n}\rangle$ and $s_a^\alpha \notin R(s_a^{i_a})$, $\alpha < i_a$ and $s_a^\alpha \rightsquigarrow s_a^{i_a}$. However, this means $s_a^{i_a} \rightsquigarrow s_a^\alpha \rightsquigarrow s_a^{i_a} \Rightarrow s_a^{i_a} \rightsquigarrow s_a^{i_a}$, contradicting that $s_a^{i_a}$ is not useless.
2) $\exists p_b$ such that $s_b^{i_b} \not\rightsquigarrow s_a^\alpha$ — Since $s_a^\alpha \notin R(s_b^{i_b})$, there is $s_a^\gamma \in R(s_b^{i_b})$ such that $s_b^{i_b} \not\rightsquigarrow s_a^\gamma$ and $\gamma > \alpha$, otherwise s_a^α would have been choosen to be part of $R(s_b^{i_b})$.
   Let's assume that $s_a^\alpha \in R(s_b^{i_b+\epsilon_b})$. There must be $s_a^{\alpha+1}$, $\alpha + 1 \leq \gamma$, $s_b^{i_b+\epsilon_b} \rightsquigarrow s_a^{\alpha+1}$. It leads to $s_b^{i_b} \rightarrow s_b^{i_b+\epsilon_b} \rightsquigarrow s_a^{\alpha+1} \rightarrow s_a^\gamma$ contradicting $s_b^{i_b} \not\rightsquigarrow s_a^\gamma$ (Figure 4).

■

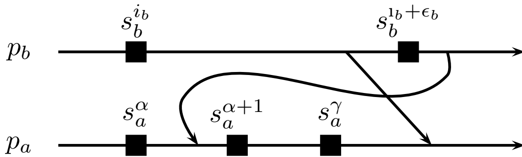

Figure 4. Second case of Lemma 1

Lemma 2. *If ${s_\rho}_a^\alpha$ is ρ-needless in a consistent cut $\mathscr{C}$ then ${s_\rho}_a^\alpha$ is erased or remains ρ-needless after a rollback in $\mathscr{C}$.*

Proof: Let $R_\rho(F)$ be the recovery line of a set of processes in $\mathscr{C}$. If $R_\rho(F)$ contains s_a^γ, $\gamma < \alpha$, ${s_\rho}_a^\alpha$ will be erased and trivially needless.

Let's consider $\gamma > \alpha$ and let's assume by contradiciton that it exists p_j such that after the rollback ${s_\rho}_a^\alpha \in R_\rho(p_j)$ in $R_\rho(F)$. Let ${s_\rho}_j^{last}$ be the last stable checkpoint of p_j in $\mathscr{C}$ and ${s_\rho}_j^{last_j} \neq {s_\rho}_j^{last}$ be the last stable checkpoint of p_j in $R_\rho(F)$, $last_j < last$.

Since ${s_\rho}_a^\gamma \notin R_\rho({s_\rho}_j^{last_j})$, ${s_\rho}_a^\gamma \rightsquigarrow {s_\rho}_j^{last_j}$. However, ${s_\rho}_a^\gamma \notin R_\rho({s_\rho}_j^{last_j}) \rightsquigarrow {s_\rho}_j^{last}$ contradicts that ${s_\rho}_a^\gamma \in R_\rho({s_\rho}_j^{last})$. ■

Theorem 2. *${s_\rho}_a^\alpha$ is ρ-needless in a consistent cut $\mathscr{C}$ if, and only if it does not belong to to any ρ-stable recovery line of $\mathscr{C}$ or any other $\mathscr{C}'$ after $\mathscr{C}$.*

Proof: ($\Rightarrow$) If ${s_\rho}_a^\alpha$ is not ρ-needless, it belongs to at least one recovery line $R_\rho(p_f)$ on $\mathscr{C}$.

($\Leftarrow$) If ${s_\rho}_a^\alpha$ is ρ-needless in $\mathscr{C}$, it does not belong to any recovery line in $\mathscr{C}$. Lemma 1 shows it will remain ρ-needless on any cut $\mathscr{C}'$, $\mathscr{C} \subseteq \mathscr{C}'$. In case of a rollback on any consistent cut $\mathscr{C}'$ after $\mathscr{C}$, Lemma 2 states that ${s_\rho}_a^\alpha$ is either erased or remains ρ-needless, again, not taking part of any recovery line. ■

V. Diskless checkpointing for RDT checkpointing protocols

A. Online garbage collection for diskless checkpointing

We can detect ρ-needless checkpoints during the distributed execution and erase them from the processes' memory with no need to use a synchronization barrier.

Definition 13. *A ρ-stable cut of a consistent cut $\mathscr{C}$ (or $\mathscr{C}_\rho$) is defined as $\mathscr{C}_\rho = h_\rho(p_1) \cup h_\rho(p_2) \cup \ldots h_\rho(p_n)$.*

Theorem 3. *If s_a^α is needless in a consistent cut $\mathscr{C}$, then s_a^α is ρ-needless in any consistent cut $\mathscr{C}'$ where ${\mathscr{C}_\rho}' \supseteq \mathscr{C}$.*

Proof: Consider $LR = \{R(s_1^{last}), \ldots, R(s_n^{last})\}$ on $\mathscr{C}$. Since s_a^α is stable and needless, it does not belong to any recovery line on LR and belongs to the cut $\langle s_1^{last}, \ldots, s_n^{last}\rangle$. Therefore, by Lemma 1, there is no cut $\langle s_1^{last+\epsilon_1}, \ldots, s_n^{last+\epsilon_n}\rangle, \epsilon_1, \ldots, \epsilon_n \geq 0$ such that s_a^α is part of a recovery line.

We need to show that s_a^α is ρ-stable and does not belong to any ρ-stable recovery line at $\mathscr{C}'$. Given s_a^{last} the last stable checkpoint at cut $\mathscr{C}$ and $s_a^{last_\rho}$ the last ρ-stable checkpoint at cut $\mathscr{C}'$, $last \leq last_\rho$. Therefore, $\langle {s_\rho}_1^{last_\rho}, \ldots, {s_\rho}_n^{last_\rho}\rangle = \langle s_1^{last+\epsilon_1}, \ldots, s_n^{last+\epsilon_n}\rangle, \epsilon_1, \ldots, \epsilon_n \geq 0$, and s_a^α is not part of any recovery line of this cut. Since $\langle {s_\rho}_1^{last_\rho}, \ldots, {s_\rho}_n^{last_\rho}\rangle$ is used to define the ρ-stable recovery lines of $\mathscr{C}'$, we can conclude that s_a^α is ρ-needless on $\mathscr{C}'$. ■

Theorem 3 can be rewritten using some properties of RDT checkpointing protocols. On them, given a causal dependency vector DV of a checkpoint , the global state $\langle c_1^{DV(c_a^\alpha)[1]}, \ldots, c_n^{DV(c_a^\alpha)[n]}\rangle$ is consistent. [8].

Corollary 4. *Consider, for any process* p_a, $last_\rho(p_a)$ *the most recent* ρ*-stable checkpoint and* $DV_\rho(s_a^\alpha)$ *the dependency vector at the cut* $\mathscr{C}$ *where* s_a^α *became needless. Given* checkpoint s_a^α *a needless checkpoint at a consistent cut* $\mathscr{C}$ *where* $DV_\rho(s_a^\alpha) = DV(v_a)$, *then* s_a^α *is* ρ*-needless in any consistent cut* $\mathscr{C}'$ *where* $\forall i, last_\rho(i) \geq DV_\rho(s_a^\alpha)[i]$.

Proof: From Theorem 3 and from $\langle s_1^{DV_\rho(s_a^\alpha)[1]}, \ldots, s_n^{DV_\rho(s_a^\alpha)[n]} \rangle$ being a consistent global checkpoint [8]. ■

However, we must determine: (i) the most recent ρ-stable checkpoint of every process and (ii) the exact cut where s_a^α became needless. We will assume each process knows its most recent ρ-stable checkpoint (for example, by receiving a reply message from the processes responsible for storing packages from a particular checkpoint). So, each process has a vector DV_ρ which will be updated the following way:

- During initialization: $\forall i, DV_\rho[i] \leftarrow 0$
- After updating `last_ρ`: $DV_\rho[pid] \leftarrow$ last_ρ
- After receiving a message m.$DV_\rho[i]$: $\forall i, DV_\rho[i] \leftarrow max\{DV_\rho[i], m.DV_\rho[i]\}$

Therefore, vector DV_ρ of p_a contains the last ρ-stable checkpoint of each process known by this process. Corollary 4 shows that if $\forall i, DV_\rho[i] \geq DV_\rho(s_a^\alpha)[i]$ in a consistent cut $\mathscr{C}$, then s_a^α is ρ-needless.

The last step is to determine the exact consistent cut when a checkpoint became needless. In [9] an online checkpoint garbage collection algorithm based on an RDT protocol tracks needless checkpoints during protocol execution, and can be used to get the dependency vector of the current state where a checkpoint became needless. Using this as DV_ρ we now have all the necessary information to characterize a ρ-needless checkpoint during the distributed execution, and it will be the base of our diskless checkpoint protocol.

B. RDT-Diskless

Algorithm 1 describes RDT-Diskless, a quasi-synchronous diskless checkpointing protocol with garbage collection. Although it is not possible to get a upper limit on the number of checkpoints, this protocol achieve checkpoint replication and garbage collection with no need of synchronization.

Type `cinfo` contains information about whether a checkpoint is currently distributed among repositories or not. Besides `dv` and `dvrho`, described previously, a process contains a vector `cm` with information about non-needless checkpoints and a set `nl`, with information about needless but not ρ-needless checkpoints.

Four auxiliary functions were created to help keeping track of the current state of each protocol:

- **new**: Records information about a new checkpoint.
- **link**: Tracks a new dependency for a non-needless checkpoints.
- **free**: Moves information about a needless checkpoint from `cm` to `nl` and stores its `dvrho` data.
- **garbage**: Checks if there is any needless checkpoint which became ρ-needless.

Each process is responsible for keeping track of its own packages, checking whether they are ρ-stable or not (using, for example, a confirmation protocol). Each process starts taking an initial checkpoint and sending packages, waiting for confirmation before execute the application itself. Every time a new checkpoint is stored a new `cinfo` register is created and tracks all dependencies related to it, allowing RDT-Diskless to determine the cut where it becomes needless.

Vectors `dv` and `dvrho` are piggbacked on every message. When a process receives a message, it updates its own vectors, then checks if any checkpoint had became needless and the dependency vector where it ocurred. All information about recently needless checkpoint is then removed from `cm` and moved to `nl`, waiting for the progress of $\mathscr{C}_\rho$ measured by `dvrho`. Every time `dvrho` is updated function `garbage` is called and checks if a checkpoint became ρ-needless and, if that happened, sends an order to the repositores to erase it.

VI. *Cheops*: A TESTING FRAMEWORK FOR DISKLESS AND DISK-BASED CHECKPOINTING PROTOCOLS

Cheops[2] provides a flexible environment necessary to compare several different implementations of both diskless and disk-based checkpointing protocols. It was implemented entirely on Python using only basic resources provided by the language, with no other auxiliary communication framework.

All communication channels are TCP channels between each pair of processes. After the creation of the communication channels (coordinated by a `dispatcher` process), the processes can only communicate by exchanging messages. Checkpoints can be stored on a filesystem, in memory or using Amazon Simple Storage Service (S3). For these tests we used a token ring application with a large CPU payload after. Figure 5 shows the measured times 16 nodes representing the mean and standard errors obtained. All tests were executed at Amazon Elastic Cloud Computing (EC2) using standard Ubuntu 9.10 servers with all packages provided by Ubuntu standard repositories.

[2]http://cheops.googlecode.com, available under Apache License

Algorithm 1 RDT-REP

Type cinfo:
ind: int
rc: int
dvrho: array [1..N] of int

Process' variables:
dv: array [1..N] of int
dvrho: array [1..N] of int
cm: array [1..N] of *cinfo
nl: set of *cinfo

Process' initialization:
$\forall i$: dv[i] $\leftarrow$ 0
$\forall i$: dvrho[i] $\leftarrow$ 0
$\forall i$: cm[i] $\leftarrow$ **null**
nl $\leftarrow \emptyset$
Take ρ-stable checkpoint $s_{\rho_i}^0$

Send *m* to p_s:
$\forall i$: m.dv[i] $\leftarrow$ dv[i]
$\forall i$: m.dvrho[i] $\leftarrow$ dvrho[i]
Transmit *m* to p_s

Receive *m* from p_s:
$\forall i$: last_dv[i] $\leftarrow$ dv[i]
$\forall i$: dv[i] $\leftarrow$ max{dv[i], m.dv[i]}
$\forall i$: dvrho[i] $\leftarrow$ max{dvrho[i], m.dvrho[i]}
$\forall i$: if (dv[i] > last_dv[i]):
free(i)
link (i, pid)
garbage()
deliver m to application

Take new checkpoint:
Send a package to every repository of p_{pid}
free(pid)
new(pid, dv[pid])
garbage()
dv[pid] $\leftarrow$ dv[pid] + 1

When c_{pid}^{α} is ρ-stable:
dvrho[pid] $\leftarrow$ max{dvrho[pid],α}
garbage()

new(i, idx: int):
cm[i] $\leftarrow$ new cinfo
*cm[i].ind $\leftarrow$ ind
*cm[i].rc $\leftarrow$ 1

link(i,j: int):
cm[i] $\leftarrow$ cm[j]
*cm[i].rc $\leftarrow$ *cm[i].rc + 1

free(i: int):
if (cm[i] $\neq$ **null**)
*cm[i].rc $\leftarrow$ *cm[i].rc - 1
if (*cm[i].rc = 0)
$\forall j$: *cm[i].dvrho[j] = dvr[j]
nl $\leftarrow$ nl $\cup$ {cm[i]}
cm[i] $\leftarrow$ **null**

garbage():
$\forall k \in nl$:
if ($\forall i$: dvrho[i] $\geq$ *k.dvrho[i])
nl $\leftarrow$ nl $-\{k\}$
*remove *k.ind from repositories*

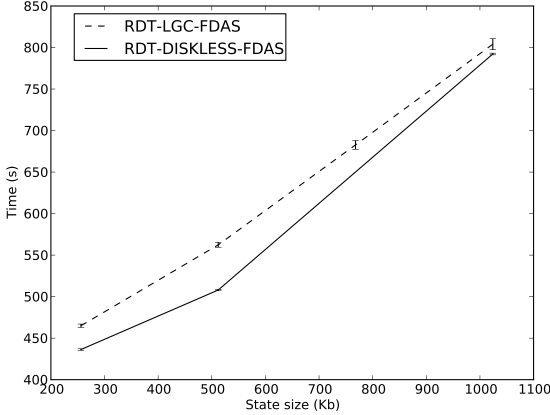

Figure 5. Measured times for 16 nodes

During these tests we compared the performance of two different quasi-synchronous RDT protocols: an FDAS coupled with RDT-LGC garbage collection [9], using S3 as a centralized repository, and FDAS coupled with RDT-Diskless diskless online garbage collection, storing checkpoints in memory and using replication (2 packages) as (ρ, ϕ)-algorithm. Every experiment was executed several times (between 6 and 10) for each checkpoint size and each process had its time recorded individually, obtaining an average of 100 measured times for each point on chart.

VII. Conclusion and Future Work

This article presented a new approach for diskless checkpointing, based on quasi-synchronous protocols and distributed in-memory storage instead of either having a synchronization barrier or a unique checkpoint repository. Moreover, an open framework was developed to compare performance of some protocols. The results obtained showed the proposed protocol has a performance improvement over a centralized approach.

Using Amazon Web Services (namely EC2 and S3) allowed us to have a customized set of machines on a high performance network on a surprisingly low cost. *Cheops* was designed to be simple and open source, letting people reproduce our results on the same platform or on its own cluster. With larger clusters available with a relatively low cost, testing on these platforms will become more and more relevant, and even with a small set of machines and protocols we obtained a better performance using a distributed storage approach. We expect that over time centralized approachs would become really inpractical due to excessive network infrastructure required to exchange all control messages and data, favoring in-memory storage.

After that we intend to analyze others partition algorithms and different checkpoint protocols, combining them properly and, using *Cheops*, profiling the application and measuring its performance. This way, we expect to create a thorough study on diskless and disk-based checkpoints, including our novel approach.

REFERENCES

[1] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon, "The TOP 500 supercomputing sites," 2009. [Online]. Available: http://www.top500.org

[2] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich *et al.*, "An overview of the BlueGene/L supercomputer," in *Supercomputing, ACM/IEEE 2002 Conference*, 2002, pp. 60–60.

[3] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *IEEE micro*, vol. 23, no. 2, pp. 22–28, 2003.

[4] Z. Chen and J. Dongarra, "A Scalable Checkpoint Encoding Algorithm for Diskless Checkpointing," in *11th IEEE High Assurance Systems Engineering Symposium, 2008. HASE 2008*, 2008, pp. 71–79.

[5] M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computing Systems (TCOS)*, vol. 3, no. 1, pp. 63–75, 1985.

[6] N. Kofahi, S. Al-Bokhitan, and A. Al-Nazer, "On Disk-based and Diskless Checkpointing for Parallel and Distributed Systems: An Empirical Analysis," *Information Technology Journal*, vol. 4, no. 4, pp. 367–376, 2005.

[7] J. S. Plank, K. Li, and M. A. Puening, "Diskless checkpointing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 10, p. 972, 1998.

[8] Y. M. Wang, "Consistent global checkpoints that contain a given set of local checkpoints," *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 456–468, 1997.

[9] R. Schmidt, I. C. Garcia, F. Pedone, and L. E. Buzato, "Optimal asynchronous garbage collection for rdt checkpointing protocols," in *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

[10] J. S. Plank and K. Li, "Faster checkpointing with n + 1 parity," University of Tennessee, Knoxville, TN, USA, Tech. Rep., 1993.

[11] S. Wicker and V. Bhargava, *Reed-Solomon codes and their applications*. Wiley-IEEE Press, 1999.

[12] M. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM (JACM)*, vol. 36, no. 2, pp. 335–348, 1989.

[13] C. Engelmann and A. Geist, "A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform," in *Challenges of Large Applications in Distributed Environments, 2003. Proceedings of the International Workshop on*, 2003, pp. 47–52.

[14] J. Plank, Y. Kim, and J. Dongarra, "Algorithm-based diskless checkpointing for fault tolerant matrix operations," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, 1995, pp. 351–360.

[15] C. Lu, "Scalable diskless checkpointing for large parallel systems," Ph.D. dissertation, University of Illinois, 2005.

[16] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Fault tolerant high performance computing by a coding approach," in *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM Press, 2005, pp. 213–223.

[17] J. Chiu and W. Hao, "Mutual-Aid: Diskless Checkpointing Scheme for Tolerating Double Faults," in *10th IEEE International Conference on High Performance Computing and Communications, 2008. HPCC'08*, 2008, pp. 540–547.

[18] L. Silva and J. Silva, "Using two-level stable storage for efficient checkpointing," *Software, IEE Proceedings*, vol. 145, pp. 198–201, 1998.

[19] P. Sobe, "Stable checkpointing in distributed systems without shared disks," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, 2003, pp. 8 pp.–.

[20] E. N. Elnozahy, D. Johnson, and Y.M.Yang, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.

[21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[22] I. C. Garcia and L. E. Buzato, "Progressive construction of consistent global checkpoints," in *19th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1999.

[23] R. H. B. Netzer and J. Xu, "Necessary and sufficient conditions for consistent global snapshots," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 165–169, 1995.

[24] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. Fuchs, "Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 5, pp. 546–554, 1995.

[25] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: transparent checkpointing under unix," in *USENIX 1995 Technical Conference Proceedings*. USENIX Association, 1995.